51CTO学院 系列丛书

Broadview®
www.broadview.com.cn



实战Java虚拟机

JVM故障诊断与性能优化

葛一鸣 | 著

通过200余示例详解Java虚拟机的各种参数配置、故障排查、性能监控及优化
技术全面，通俗易懂

 中国工信出版集团  電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

LANZAROTE Caliente.COM

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版



实战Java虚拟机

JVM故障诊断与性能优化

对Java程序员来说，Java虚拟机（JVM）可以说是既熟悉又神秘，很少有Java程序员能够抑制自己探究它的冲动。可惜分析JVM故障诊断与性能优化的书籍（尤其是国内出版的）简直少之又少。本书的出版可谓研究JVM的程序员的福音，作者注重理论联系实际，对于理论性较强的章节和知识点安排了大量的实践案例来说明和进行实际操作，具有非常强的实践指导意义。同时本书配套操作视频《深入浅出Java虚拟机——入门篇》在51CTO学院独家发布后受到了51CTO社区广大开发者和爱好者的好评和认可，所以，强烈推荐本书给爱好JVM的你！

——51CTO学院高级运营经理 曹亚莉



博文视点Broadview




@博文视点Broadview

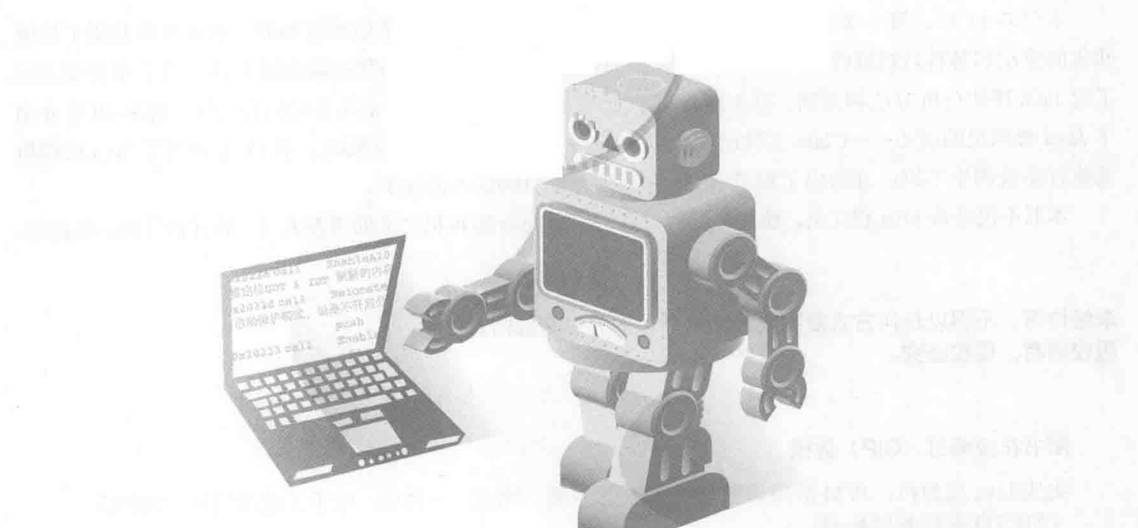

责任编辑：董 英
封面设计：李 玲

上架建议：程序设计>Java


ISBN 978-7-121-25612-7

定价：79.00元

51CTO学院系列丛书



实战Java虚拟机

JVM故障诊断与性能优化

葛一鸣 | 著

電子工業出版社
Publishing House of Electronics Industry

内 容 简 介

随着越来越多的第三方语言（Groovy、Scala、JRuby 等）在 Java 虚拟机上运行，Java 也俨然成为了一个充满活力的生态圈。本书将通过 200 余示例详细介绍 Java 虚拟机中的各种参数配置、故障排查、性能监控以及性能优化。

本书共 11 章。第 1~3 章介绍了 Java 虚拟机的定义、总体架构、常用配置参数。第 4~5 章介绍了垃圾回收的算法和各种垃圾回收器。第 6 章介绍了 Java 虚拟机的性能监控和故障诊断工具。第 7 章详细介绍了对 Java 堆的分析方法和案例。第 8 章介绍了 Java 虚拟机对多线程，尤其是对锁的支持。第 9~10 章介绍了 Java 虚拟机的核心——Class 文件结构，以及 Java 虚拟机中类的装载系统。第 11 章介绍了 Java 虚拟机的执行系统和字节码，并给出了通过 ASM 框架进行字节码注入的案例。

本书不仅适合 Java 程序员，还适合任何一名工作于 Java 虚拟机之上的研发人员、软件设计师、架构师。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

实战 Java 虚拟机：JVM 故障诊断与性能优化 / 葛一鸣著. 一北京：电子工业出版社，2015.3
（51CTO 学院系列丛书）
ISBN 978-7-121-25612-7

Ⅰ. ①实… Ⅱ. ①葛… Ⅲ. ①JAVA 语言一程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 040500 号

责任编辑：董　英
印　　刷：北京中新伟业印刷有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：28.25　字数：633 千字
版　　次：2015 年 3 月第 1 版
印　　次：2015 年 3 月第 1 次印刷
印　　数：3000 册　定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

关于 Java 生态圈

Java 是目前应用最为广泛的软件开发平台之一。随着 Java 以及 Java 社区的不断壮大，Java 也早已不再是简简单单的一门计算机语言了，它更是一个平台、一种文化、一个社区。

作为一个平台，Java 虚拟机扮演着举足轻重的作用。除了 Java 语言，任何一种能够被编译成字节码的计算机语言都属于 Java 这个平台。Groovy、Scala、JRuby 等都是 Java 平台的一个部分，它们依赖于 Java 虚拟机，同时，Java 平台也因为它们变得更加丰富多彩。

作为一种文化，Java 几乎成为了“开源”的代名词。在 Java 程序中，有着数不清的开源软件和框架，如 Tomcat、Struts、Hibernate、Spring 等。就连 JDK 和 JVM 自身也有不少开源的实现，如 OpenJDK、Harmony。可以说，“共享”的精神在 Java 世界里体现得淋漓尽致。

作为一个社区，Java 拥有无数的开发人员，有数不清的论坛和资料。从桌面应用软件、嵌入式开发到企业级应用、后台服务器、中间件，都可以看到 Java 的身影。其应用形式之复杂、参与人数之众多也令人咋舌。可以说，Java 社区已经俨然成为了一个良好而庞大的生态系统。

而本书，将主要介绍这个生态系统的核心——**Java 虚拟机**。

本书的体系结构

本书立足于实际开发，又不缺乏理论介绍，力求通俗易懂、循序渐进。本书共分为 11 章：

第 1 章主要为综述，介绍了 Java 虚拟机的概念、定义，讲解了 Java 语言规范和 Java 虚拟机规范，最后，还介绍了 OpenJDK 的调试方法。

第 2 章介绍了 Java 虚拟机的总体架构，说明了堆、栈、方法区等内存空间的作用和彼此之间的联系。

第 3 章介绍了 Java 虚拟机的常用配置参数，重点对垃圾回收跟踪参数、内存配置参数做了详细的介绍，并给出了案例说明。

第 4 章从理论层面介绍了垃圾回收的算法，如引用计数、标记清除、标记压缩、复制算法等。本章是第 5 章的理论基础。

第 5 章讲解了基于垃圾回收的理论知识，进一步详细介绍了 Java 虚拟机中实际使用的各种垃圾回收器，包括串行回收器、并行回收器、CMS、G1 等。

第 6 章介绍了 Java 虚拟机的性能监控和故障诊断工具，考虑到实用性，也介绍了系统级性能监控工具的使用，两者结合，可以更好地帮助读者处理实际问题。

第 7 章详细介绍了对 Java 堆的分析方法和案例，主要讲解了 MAT 和 Visual VM 两款工具的使用，以及各自 OQL 的编写方式。

第 8 章介绍了 Java 虚拟机对多线程，尤其是对锁的支持，本章不仅介绍了虚拟机内部锁的实现、优化机制，也给出了一些 Java 语言层面的锁优化思路，最后，还介绍了无锁的并行控制方法。

第 9 章介绍了 Java 虚拟机的核心——Class 文件结构，Class 文件作为 Java 虚拟机的基石，有着举足轻重的作用，对深入理解 Java 虚拟机有着不可忽视的作用。

第 10 章介绍了 Java 虚拟机中类的装载系统，其中，着重介绍了 Java 虚拟机中 ClassLoader 的实现以及设计模式。

第 11 章介绍了 Java 虚拟机的执行系统和字节码，为了帮助读者更快更好地理解 Java 字节码，本章对字节码进行了分类讲解，并且理论联系实际，给出了通过 ASM 框架进行字节码注入的案例。

本书特色

本书的主要特点有：

1. **结构清晰**。本书采用从整体到局部的视角，首先第 1、2 章介绍了 Java 虚拟机的整体概况和结构。接着步步为营，每一章节对应一个单独的知识点，力求展示虚拟机的全貌。

2. **理论结合实战**。本书不甘心于简单地枚举理论知识，在每一个理论背后，都给出了演示示例供读者参考，帮助读者更好地消化这些理论。比如，在对 Class 文件结构和字节码的介绍中，不仅仅简单地给出了理论说明，更是使用 ASM 框架将这些理论应用于实践，尽可能地做到理论和实践结合。
3. **专注专业**。本书着眼于 Java 虚拟机，对 Java 虚拟机的原理和实践做了丰富的介绍，包括但不限于体系结构、虚拟机的调试方式、常用参数、垃圾回收系统、Class 文件结构、执行系统等，力求从多角度更专业地对 Java 虚拟机进行探讨。
4. **通俗易懂**。本书依然服务于广大虚拟机初学者，尽量避免采用过于理论的描述方式，简单的白话文风格贯穿全书，尽量做到读者在阅读过程中少盲点、无盲点。
5. **技术全面**。纵横 Windows 和 Linux 双系统下的性能诊断、涉及 32 位系统和 64 位系统的优化比较、贯穿从 JDK 1.5 到 JDK 1.8 的优化演进。

适合阅读人群

虽然本书力求通俗，但要通读本书并取得良好的学习效果，要求读者需要具备基本的 Java 知识或者一定的编程经验。因此，本书适合以下读者：

- 拥有一定开发经验的 Java 平台开发人员（Java、Scala、JRuby 等）
- 软件设计师、架构师
- 系统调优人员
- 有一定的 Java 编程基础并希望进一步理解 Java 的程序员
- 虚拟机爱好者，JVM 实践者

本书的约定

本书在叙述过程中，有如下约定：

- 本书中所述的 JDK 1.5、JDK 1.6、JDK 1.7、JDK 1.8 等同于 JDK 5、JDK 6、JDK 7、JDK 8。
- 如无特殊说明，Java 虚拟机均指 HotSpot 虚拟机。
- 如无特殊说明，本书的程序、示例均在 JDK 1.7 环境中运行。
- 本书赠送的课程优惠券，可以观看笔者在 51CTO 学院的 JVM 课程，地址是 http://edu.51cto.com/course/course_id-1952.html。

联系作者

本书的写作过程远比我想象的更艰辛，为了让全书能够更清楚、更正确地表达和论述，我经历了好多个不眠之夜，即使现在回想起来，也忍不住让我打个寒战。由于写作水平的限制，书中难免会有不妥之处，望读者谅解。

为此，如果读者有任何疑问或者建议，非常欢迎大家加入 QQ 群 397196583 ，一起探讨学习中的困难、分享学习的经验，我期待与大家一起交流、共同进步。同时，也希望大家可以关注我的博客 http://www.uucode.net/。

感谢

这本书能够面世，是因为得到了众人的支持。首先，要感谢我的妻子，她始终不辞辛劳，毫无怨言地对我照顾有加，才让我得以腾出大量时间，并可以安心工作。其次，要感谢小编为我一次又一次地审稿改错，批评指正，才能让本书逐步完善。最后，感谢我的母亲 30 年如一日对我的体贴和关心。

参与本书编写的还有宋玉红、关硕、安继宏、白慧、薛淑英、蒋玺、曹静、马玉杰、陈明明、张丽萍、任娜娜、李清艺、荆海霞、赵全利、孙迪，特此感谢！

葛一鸣

目　录

第 1 章　初探 Java 虚拟机 1
1.1　知根知底：追溯 Java 的发展历程 2
1.1.1　那些依托 Java 虚拟机的语言大咖们 2
1.1.2　Java 发展史上的里程碑 2
1.2　跨平台的真相：Java 虚拟机来做中介 4
1.2.1　理解 Java 虚拟机的原理 4
1.2.2　看清 Java 虚拟机的种类 5
1.3　一切看我的：Java 语言规范 6
1.3.1　词法的定义 6
1.3.2　语法的定义 7
1.3.3　数据类型的定义 8
1.3.4　Java 语言规范总结 9
1.4　一切听我的：Java 虚拟机规范 9
1.5　数字编码就是计算机世界的水和电 10
1.5.1　整数在 Java 虚拟机中的表示 10
1.5.2　浮点数在 Java 虚拟机中的表示 12
1.6　抛砖引玉：编译和调试虚拟机 14
1.7　小结 19

第 2 章　认识 Java 虚拟机的基本结构 20
2.1　谋全局者才能成大器：看穿 Java 虚拟机的架构 20

2.2 小参数能解决大问题：学会设置 Java 虚拟机的参数......22
2.3 对象去哪儿：辨清 Java 堆......23
2.4 函数如何调用：出入 Java 栈......25
2.4.1 局部变量表......27
2.4.2 操作数栈......32
2.4.3 帧数据区......32
2.4.4 栈上分配......33
2.5 类去哪儿了：识别方法区......35
2.6 小结......37

第 3 章 常用 Java 虚拟机参数......38
3.1 一切运行都有迹可循：掌握跟踪调试参数......38
3.1.1 跟踪垃圾回收——读懂虚拟机日志......39
3.1.2 类加载/卸载的跟踪......42
3.1.3 系统参数查看......44
3.2 让性能飞起来：学习堆的配置参数......45
3.2.1 最大堆和初始堆的设置......45
3.2.2 新生代的配置......49
3.2.3 堆溢出处理......52
3.3 别让性能有缺口：了解非堆内存的参数配置......54
3.3.1 方法区配置......55
3.3.2 栈配置......55
3.3.3 直接内存配置......55
3.4 Client 和 Server 二选一：虚拟机的工作模式......58
3.5 小结......59

第 4 章 垃圾回收概念与算法......60
4.1 内存管理清洁工：认识垃圾回收......60
4.2 清洁工具大 PK：讨论常用的垃圾回收算法......61
4.2.1 引用计数法（Reference Counting）......62
4.2.2 标记清除法（Mark-Sweep）......63
4.2.3 复制算法（Copying）......64

4.2.4 标记压缩法（Mark-Compact）......66
4.2.5 分代算法（Generational Collecting）......67
4.2.6 分区算法（Region）......68
4.3 谁才是真正的垃圾：判断可触及性......69
4.3.1 对象的复活......69
4.3.2 引用和可触及性的强度......71
4.3.3 软引用——可被回收的引用......72
4.3.4 弱引用——发现即回收......76
4.3.5 虚引用——对象回收跟踪......77
4.4 垃圾回收时的停顿现象：Stop-The-World 案例实战......79
4.5 小结......83

第 5 章 垃圾收集器和内存分配......84

5.1 一心一意一件事：串行回收器......85
5.1.1 新生代串行回收器......85
5.1.2 老年代串行回收器......86
5.2 人多力量大：并行回收器......86
5.2.1 新生代 ParNew 回收器......87
5.2.2 新生代 ParallelGC 回收器......88
5.2.3 老年代 ParallelOldGC 回收器......89
5.3 一心多用都不落下：CMS 回收器......90
5.3.1 CMS 主要工作步骤......90
5.3.2 CMS 主要的设置参数......91
5.3.3 CMS 的日志分析......92
5.3.4 有关 Class 的回收......94
5.4 未来我做主：G1 回收器......95
5.4.1 G1 的内存划分和主要收集过程......95
5.4.2 G1 的新生代 GC......96
5.4.3 G1 的并发标记周期......97
5.4.4 混合回收......100
5.4.5 必要时的 Full GC......102
5.4.6 G1 日志......102

5.4.7 G1 相关的参数 106
5.5 回眸：有关对象内存分配和回收的一些细节问题 107
5.5.1 禁用 System.gc() 107
5.5.2 System.gc()使用并发回收 107
5.5.3 并行 GC 前额外触发的新生代 GC 109
5.5.4 对象何时进入老年代 110
5.5.5 在 TLAB 上分配对象 117
5.5.6 方法 finalize()对垃圾回收的影响 120
5.6 温故又知新：常用的 GC 参数 125
5.7 动手才是真英雄：垃圾回收器对 Tomcat 性能影响的实验 127
5.7.1 配置实验环境 127
5.7.2 配置进行性能测试的工具 JMeter 128
5.7.3 配置 Web 应用服务器 Tomcat 131
5.7.4 实战案例 1——初试串行回收器 133
5.7.5 实战案例 2——扩大堆以提升系统性能 133
5.7.6 实战案例 3——调整初始堆大小 134
5.7.7 实战案例 4——使用 ParrellOldGC 回收器 135
5.7.8 实战案例 5——使用较小堆提高 GC 压力 135
5.7.9 实战案例 6——测试 ParallelOldGC 的表现 135
5.7.10 实战案例 7——测试 ParNew 回收器的表现 136
5.7.11 实战案例 8——测试 JDK 1.6 的表现 136
5.7.12 实战案例 9——使用高版本虚拟机提升性能 137
5.8 小结 137

第 6 章 性能监控工具 138
6.1 有我更高效：Linux 下的性能监控工具 139
6.1.1 显示系统整体资源使用情况——top 命令 139
6.1.2 监控内存和 CPU——vmstat 命令 140
6.1.3 监控 IO 使用——iostat 命令 142
6.1.4 多功能诊断器——pidstat 工具 143
6.2 用我更高效：Windows 下的性能监控工具 148
6.2.1 任务管理器 148

6.2.2 perfmon 性能监控工具 150
6.2.3 Process Explorer 进程管理工具 153
6.2.4 pslist 命令——Windows 下也有命令行工具 155
6.3 外科手术刀：JDK 性能监控工具 157
6.3.1 查看 Java 进程——jps 命令 158
6.3.2 查看虚拟机运行时信息——jstat 命令 159
6.3.3 查看虚拟机参数——jinfo 命令 162
6.3.4 导出堆到文件——jmap 命令 163
6.3.5 JDK 自带的堆分析工具——jhat 命令 165
6.3.6 查看线程堆栈——jstack 命令 167
6.3.7 远程主机信息收集——jstatd 命令 170
6.3.8 多功能命令行——jcmd 命令 172
6.3.9 性能统计工具——hprof 175
6.3.10 扩展 jps 命令 177
6.4 我是你的眼：图形化虚拟机监控工具 JConsole 178
6.4.1 JConsole 连接 Java 程序 178
6.4.2 Java 程序概况 179
6.4.3 内存监控 180
6.4.4 线程监控 180
6.4.5 类加载情况 182
6.4.6 虚拟机信息 182
6.5 一目了然：可视化性能监控工具 Visual VM 183
6.5.1 Visual VM 连接应用程序 184
6.5.2 监控应用程序概况 185
6.5.3 Thread Dump 和分析 186
6.5.4 性能分析 187
6.5.5 内存快照分析 189
6.5.6 BTrace 介绍 190
6.6 来自 JRockit 的礼物：虚拟机诊断工具 Mission Control 198
6.6.1 MBean 服务器 198
6.6.2 飞机记录器（Flight Recorder） 200
6.7 小结 203

第 7 章 分析 Java 堆......204
7.1 对症才能下药：找到内存溢出的原因......205
7.1.1 堆溢出......205
7.1.2 直接内存溢出......205
7.1.3 过多线程导致 OOM......207
7.1.4 永久区溢出......209
7.1.5 GC 效率低下引起的 OOM......210
7.2 无处不在的字符串：String 在虚拟机中的实现......210
7.2.1 String 对象的特点......210
7.2.2 有关 String 的内存泄漏......212
7.2.3 有关 String 常量池的位置......215
7.3 虚拟机也有内窥镜：使用 MAT 分析 Java 堆......217
7.3.1 初识 MAT......217
7.3.2 浅堆和深堆......220
7.3.3 例解 MAT 堆分析......221
7.3.4 支配树（Dominator Tree）......225
7.3.5 Tomcat 堆溢出分析......226
7.4 筛选堆对象：MAT 对 OQL 的支持......230
7.4.1 Select 子句......230
7.4.2 From 子句......232
7.4.3 Where 子句......234
7.4.4 内置对象与方法......234
7.5 更精彩的查找：Visual VM 对 OQL 的支持......239
7.5.1 Visual VM 的 OQL 基本语法......239
7.5.2 内置 heap 对象......240
7.5.3 对象函数......242
7.5.4 集合/统计函数......247
7.5.5 程序化 OQL 分析 Tomcat 堆......252
7.6 小结......255

第 8 章 锁与并发......256
8.1 安全就是锁存在的理由：锁的基本概念和实现......257

8.1.1 理解线程安全......257
8.1.2 对象头和锁......259
8.2 避免残酷的竞争：锁在 Java 虚拟机中的实现和优化......260
8.2.1 偏向锁......260
8.2.2 轻量级锁......262
8.2.3 锁膨胀......263
8.2.4 自旋锁......264
8.2.5 锁消除......264
8.3 应对残酷的竞争：锁在应用层的优化思路......266
8.3.1 减少锁持有时间......266
8.3.2 减小锁粒度......267
8.3.3 锁分离......269
8.3.4 锁粗化......271
8.4 无招胜有招：无锁......273
8.4.1 理解 CAS......273
8.4.2 原子操作......274
8.4.3 新宠儿 LongAddr......277
8.5 将随机变为可控：理解 Java 内存模型......280
8.5.1 原子性......280
8.5.2 有序性......282
8.5.3 可见性......284
8.5.4 Happens-Before 原则......286
8.6 小结......286

第 9 章 Class 文件结构......287
9.1 不仅跨平台，还能跨语言：语言无关性......287
9.2 虚拟机的基石：Class 文件......289
9.2.1 Class 文件的标志——魔数......290
9.2.2 Class 文件的版本......292
9.2.3 存放所有常数——常量池......293
9.2.4 Class 的访问标记（Access Flag）......300
9.2.5 当前类、父类和接口......301

9.2.6 Class 文件的字段 302
9.2.7 Class 文件的方法基本结构 304
9.2.8 方法的执行主体——Code 属性 306
9.2.9 记录行号——LineNumberTable 属性 307
9.2.10 保存局部变量和参数——LocalVariableTable 属性 308
9.2.11 加快字节码校验——StackMapTable 属性 308
9.2.12 Code 属性总结 313
9.2.13 抛出异常——Exceptions 属性 314
9.2.14 用实例分析 Class 的方法结构 315
9.2.15 我来自哪里——SourceFile 属性 318
9.2.16 强大的动态调用——BootstrapMethods 属性 319
9.2.17 内部类——InnerClasses 属性 320
9.2.18 将要废弃的通知——Deprecated 属性 321
9.2.19 Class 文件总结 322
9.3 操作字节码：走进 ASM 322
9.3.1 ASM 体系结构 322
9.3.2 ASM 之 Hello World 324
9.4 小结 325

第 10 章 Class 装载系统 326

10.1 来去都有序：看懂 Class 文件的装载流程 326
10.1.1 类装载的条件 327
10.1.2 加载类 330
10.1.3 验证类 332
10.1.4 准备 333
10.1.5 解析类 334
10.1.6 初始化 336
10.2 一切 Class 从这里开始：掌握 ClassLoader 340
10.2.1 认识 ClassLoader，看懂类加载 341
10.2.2 ClassLoader 的分类 341
10.2.3 ClassLoader 的双亲委托模式 343
10.2.4 双亲委托模式的弊端 347

10.2.5 双亲委托模式的补充 348
10.2.6 突破双亲模式 350
10.2.7 热替换的实现 353
10.3 小结 357

第 11 章 字节码执行 358

11.1 代码如何执行：字节码执行案例 359
11.2 执行的基础：Java 虚拟机常用指令介绍 369
11.2.1 常量入栈指令 369
11.2.2 局部变量压栈指令 370
11.2.3 出栈装入局部变量表指令 371
11.2.4 通用型操作 372
11.2.5 类型转换指令 373
11.2.6 运算指令 375
11.2.7 对象/数组操作指令 377
11.2.8 比较控制指令 379
11.2.9 函数调用与返回指令 386
11.2.10 同步控制 389
11.2.11 再看 Class 的方法结构 391
11.3 更上一层楼：再看 ASM 393
11.3.1 为类增加安全控制 393
11.3.2 统计函数执行时间 396
11.4 谁说 Java 太刻板：Java Agent 运行时修改类 399
11.4.1 使用-javaagent 参数启动 Java 虚拟机 400
11.4.2 使用 Java Agent 为函数增加计时功能 402
11.4.3 动态重转换类 404
11.4.4 有关 Java Agent 的总结 407
11.5 与时俱进：动态函数调用 407
11.5.1 方法句柄使用实例 407
11.5.2 调用点使用实例 411
11.5.3 反射和方法句柄 412
11.5.4 指令 invokedynamic 使用实例 414

11.6 跑得再快点：静态编译优化 418
11.6.1 编译时计算 419
11.6.2 变量字符串的连接 421
11.6.3 基于常量的条件语句裁剪 422
11.6.4 switch 语句的优化 423
11.7 提高虚拟机的执行效率：JIT 及其相关参数 424
11.7.1 开启 JIT 编译 425
11.7.2 JIT 编译阈值 426
11.7.3 多级编译器 427
11.7.4 OSR 栈上替换 430
11.7.5 方法内联 431
11.7.6 设置代码缓存大小 432
11.8 小结 436

第1章 初探Java虚拟机

什么是Java虚拟机？什么是Java语言？两者又有何关系？作为本书开篇之章，本章将主要介绍有关Java虚拟机的基本概念、发展历史和实现概要。其中，将重点介绍支撑Java世界的两份重要规范——Java语言规范和Java虚拟机规范，帮助读者更好地理解Java生态圈。

本章涉及的主要知识点有：

- 读懂Java的发展历史。
- 学习Java虚拟机的概念和种类。
- 接触Java语言规范。
- 了解Java虚拟机规范。
- 掌握单步调试Java虚拟机的方法。

1.1 知根知底：追溯 Java 的发展历程

目前，Java 语言可以说是最常用的编程语言之一，在应用软件领域，它唯一的竞争对手似乎只有微软的.NET。C/C++作为曾经的霸主，目前依然占据着系统软件和嵌入式系统绝对的市场份额，但正在逐步退出应用软件领域。和 C/C++相比，Java 在设计上有着绝对的优势，开发人员可以尽快从语言本身的复杂性中解脱出来，将更多的精力投向软件自身的业务功能。由于 Java 语言的这份简单性，也可以认为 Java 是一门极好的初学者入门语言。

但是，人无完人，Java 在不少地方依然受到了广大开发人员的诟病，它烦琐的语法经常受到 Python 等开发人员的耻笑。在语言的动态性上，甚至也远远不如和它年龄相仿的 PHP 语言。但为了支持动态语言，Java 虚拟机推出了新的函数调用指令 invokedynamic（本书将在第 11 章中具体介绍该指令），试图弥补 Java 在动态调用上的不足。

因此，值得欣慰，到目前为止，Java 仍然处于快速发展期，不断壮大，不断完善。

1.1.1 那些依托 Java 虚拟机的语言大咖们

无论受到多少非议，Java 的崛起已经是不争的事实。想起《康熙王朝》中的对白，哪一位千古帝王、功臣名将不是“褒满天下，谤满天下”。而且万幸的是，Java 生态系统极具活力，在 Java 8 中，已经推出了函数式编程语法，试图简化 Java 语言在语法上的诟病。如果你不喜欢这种新的语法也没关系，Clojure 语言作为 Lisp 的方言，可以很好地在 Java 虚拟机上执行。如果你受不了 Lisp 形式的怪异语法，Jython 已经可以将 Python 运行在 Java 虚拟机上。如果你只需要一个简单的脚本，Groovy 也可以成为你的选择。哦，对了，还有 Scala，专注于高并发的解决方案。在这里，你可以找到你需要的一切。

而所有这一切，仍然正在不断地蓬勃发展，它们和那个看似呆板的 Java 语言渐行渐远，但却都深深地扎根于 Java 虚拟机平台上。

1.1.2 Java 发展史上的里程碑

下面，将简要介绍一下 Java 发展史上的重大事项。

1990 年，在 Sun 计算机公司中，由 Patrick Naughton、MikeSheridan 及 James Gosling 领导的小组 Green Team，开始研发一种可控制家用电子产品的新型计算机软件技术，并希望能够研究出一种可以跨平台的系统。起先他们试着在 C++的功能基础上做修改，但一直无法克服编译器的问题，所以决定自行开发新的程序语言——Oak。这里的 Oak 已经具备安全性、网络通信、

面向对象、垃圾回收、多线程等特性。后来他们发现 Oak 已经被其他公司注册，于是改名为 Java。

1995 年，Sun 正式发布 Java 和 HotJava 产品，Java 首次公开亮相。

1996 年 1 月 23 日 Sun Microsystems 发布了 JDK 1.0。这个版本包括了两部分：运行环境（即 JRE）和开发环境（即 JDK）。在运行环境中包括了核心 API、用户界面 API、发布技术、Java 虚拟机（JVM）几个部分。开发环境包括了编译 Java 程序的编译器（即 javac）。在 JDK 1.0 时代，Java 使用一款叫作 Classic 的虚拟机解释执行 Java 字节码。

1997 年 2 月 18 日 Sun 发布了 JDK 1.1，在该版本中，已经支持 AWT、内部类、JDBC、RMI、反射等特性。同年，Sun 收购了一家叫作 Longview Technologies 的公司，从而获得了 Hotspot 虚拟机。

同在 1997 年，Jim Hugunin 创造了 Jython，但由于各种原因，Jython 的进展相当缓慢，但到现在为止，Jython 已经取得了长足的进步，甚至已经可以运行 Django 框架。

1998 年，JDK 1.2 版本发布（从这个版本开始的 Java 技术都称为 Java 2）。Java 2 不仅能兼容智能卡和小型消费类设备，还能兼容大型的服务器系统，它使软件开发商、服务提供商和设备提供商更加容易地抢占市场机遇。这一开发工具极大地简化了编程人员编制企业级 Web 应用的工作。同时 Sun 发布了 JSP/Servlet、EJB 规范，以及将 Java 分成了 J2EE、J2SE 和 J2ME。这表明了 Java 开始向企业、桌面应用和移动设备应用 3 大领域挺进。此时的 Java 已经做到解释执行和编译执行混合运行。

2000 年，JDK 1.3 发布，Hotspot 虚拟机成为 Java 的默认虚拟机。

2002 年，JDK 1.4 发布，古老的 Classic 虚拟机退出历史舞台。

2003 年年底，Java 平台的 Scala 正式发布，同年 Groovy 也加入了 Java 阵营。

2004 年，JDK 1.5 发布。同时 JDK 1.5 改名为 J2SE 5.0。在这个版本中，Java 语言做了大量的改进，比如支持了泛型、注解、自动装箱拆箱、枚举类型、可变长参数、增强的 foreach 循环等。语法上的简化和改进是这一版本的一大特色。

2006 年，JDK 1.6 发布。同年，Java 开源并建立了 OpenJDK。顺理成章，Hotspot 虚拟机也成为了 OpenJDK 中的默认虚拟机。

2007 年，Java 平台迎来了新伙伴 Clojure。

2008 年，Oracle 收购了 BEA，得到了 JRockit 虚拟机。

2009 年，Twitter 宣布把后台大部分程序从 Ruby 迁移到 Scala，这是 Java 平台的又一次大规模应用。

2010 年，Oracle 收购了 Sun，获得最具价值的 Hotspot 虚拟机。此时，Oracle 拥有市场占用率最高的两款虚拟机 Hotspot 和 JRockit，并计划在未来对它们进行整合。

2011 年，JDK 1.7 发布。在 JDK 1.7 中，正式启用了新的垃圾回收器 G1，支持了 64 位系统的压缩指针，以及 NIO 2.0，同时新增的 invokedynamic 指令也是该版本的一大特色。

2014 年，JDK 1.8 发布。在 JDK 1.8 中，全新的 Lambda 表达式是一大亮点，它彻底改变了 Java 的编程风格和习惯。

Oracle 计划在 2016 年，发布 JDK 1.9。届时，最令人期待的功能应该是 Java 的模块化。

注意：在本书中，JDK 1.6 等同于 JDK 6，JDK 1.7 等同于 JDK 7，JDK 1.8 等同于 JDK 8。

1.2 跨平台的真相：Java 虚拟机来做中介

在简单了解了 Java 的发展历程之后，本节将着重介绍 Java 虚拟机的概念，最后了解一下目前最流行的 Java 虚拟机。

1.2.1 理解 Java 虚拟机的原理

所谓虚拟机，就是一台虚拟的计算机。它是一款软件，用来执行一系列虚拟计算机指令。大体上，虚拟机可以分为系统虚拟机和程序虚拟机。大名鼎鼎的 Visual Box、VMware 就属于系统虚拟机，它们完全是对物理计算机的仿真，提供了一个可运行完整操作系统的软件平台。程序虚拟机的典型代表就是 Java 虚拟机，它专门为执行单个计算机程序而设计，在 Java 虚拟机中执行的指令我们称为 Java 字节码指令。无论是系统虚拟机还是程序虚拟机，在上面运行的软件都被限制于虚拟机提供的资源中。

图 1.1 显示了同一个 Java 程序（Java 字节码的集合），通过 Java 虚拟机运行于各大主流系统平台，该程序以虚拟机为中介，实现了跨平台的特性。

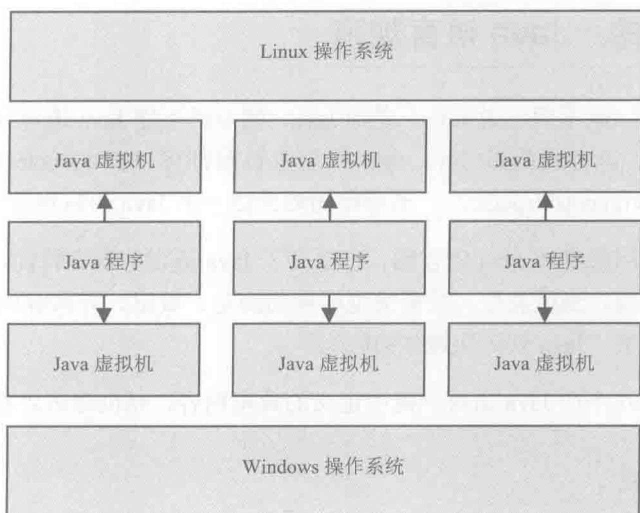


图 1.1　在操作系统之上执行的虚拟机程序

1.2.2　看清 Java 虚拟机的种类

Java 发展至今，先后出现了不少 Java 虚拟机。在 Java 发展最初，Sun 使用的是一款叫作 Classic 的 Java 虚拟机，之后，在 Solaris 平台上还曾短暂地使用过 Exact VM 虚拟机，到现在，最终被大规模部署和应用的是 Hotspot 虚拟机。

除了 Sun 公司以外，各大公司以及组织都曾积极研发过 Java 虚拟机，比如 BEA 的 JRockit，目前，JRockit 和 Hotspot 都被收入 Oracle 旗下，大有整合的趋势。在 IBM 内部，使用着一款名为 J9 的虚拟机，广泛用于 IBM 的各大产品（如果当年 IBM 成功收购了 Sun，那么很可能是 J9 和 Hotspot 进行整合了）。此外，Apache 也曾经推出过与 JDK 1.5 和 JDK 1.6 兼容的 Java 运行平台 Apache Harmony，它是开源软件，但受到同样开源的 OpenJDK 的压制，最终于 2011 年退役，虽然目前并没有 Apache Harmony 被大规模商用的案例，但是它的出现对 Android 的发展起到了极为重要的作用。在嵌入式领域，KVM 和 CDC/CLDC Hotspot 两款虚拟机也扮演着重要的角色，在 iOS 和 Android 盛行之前，这两款虚拟机也广泛运用于手机平台。

注意：由于目前 Hotspot 占有绝对的市场地位，若无特别说明，本书的示例以及参数都是针对 Hotspot 虚拟机的。

1.3 一切看我的：Java 语言规范

讲到 Java 虚拟机，就不得不说 Java，说到 Java，就不得不提 Java 语言规范（Java Language Specification）。Java 语言规范和 Java 虚拟机规范目前都可以在 Oracle 的官方网站上找到（http://docs.oracle.com/javase/specs/）。本节将简要介绍一下 Java 语言规范。

Java 语言规范是用来描述 Java 语言的，它定义了 Java 语言的语言特性，比如 Java 的语法、词法、支持的数据类型、变量类型、数据类型转换的约定、数组、异常等内容。Java 语言规范的目的是告诉开发人员“Java 代码应该如何编写”。

本节将简单介绍几个在 Java 语言规范中定义的典型内容，帮助读者理解这份规范的意图。

1.3.1 词法的定义

词法规定了 Java 语法中每一个单词如何书写才是合乎规范的。比如，词法定义中规定了 Java 的关键字，如果开发人员使用 Java 的关键字作为变量名，显然无法正常通过编译。

下面简单地看一个有关标示符的定义：

```
Identifier:
    IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

IdentifierChars:
    JavaLetter
    IdentifierChars JavaLetterOrDigit

JavaLetter:
    any Unicode character that is a Java letter

JavaLetterOrDigit:
    any Unicode character that is a Java letter-or-digit
```

这里定义了标示符由一个标示符串组成，但是不能是关键字、布尔字面量或者是 Null 字面量。而标示符串是一个由字母开头的、由字母或数字构成的串。这里的字母或数字并非简单的 ABC，而是指任意的 Unicode 字符。

根据这个定义便可以知道，下面的方法是不符合规范的：

```
public static void new (){
}
```

因为 new 为 Java 关键字，不属于标示符，因此不能用作方法名，而下述代码则是符合规范的：

```
public static void 打印(){
    System.out.println("中文方法");
}
```

虽然上述代码使用了中文作为方法名，但是根据定义，“打”、“印”属于 Unicode 字符，因此是符合规范的名字，也构成了符合规范的标示符。

有关词法另一个典型的案例就是数字的表示。在 JDK 1.7 中，以下都属于符合规范的数字：

- 整数
  - 0、2、0372 、0xDada_Cafe、1996、0x00_FF__00_FF
- 长整形
  - 0l、0777L、0x100000000L、2_147_483_648L、0xC0B0L
- 单精度浮点
  - 1e1f、2.f、 .3f、0f、3.14f、6.022137e+23f
- 双精度浮点
  - 1e1、2.、.3、0.0、3.14、1e-9d、1e137

就直观感觉而言，上述有些数字真是长得太奇怪了，比如 0xDada_Café，它不更像一个英语单词吗？而事实上，这是一个符合规范的 16 进制整数。Java 语言规范规定，0x 字符开始的表示 16 进制，同时，为了可读性，也允许在数字中间增加下画线进行分割（这是 JDK 1.7 中引入的）。

得益于 Java 语言规范对于词法的定义，现在 Java 程序可以使用更加丰富的方法来定义数字，无论是在可读性或是在表达能力上，都非常强劲。

1.3.2　语法的定义

词法定义规定了什么样的单词是合理的，语法定义规定了什么样的语句是合乎规范的。以 if 语句为例，在类似于 Basic 的语言中，可能会用以下形式定义 if 语句：

```
if Expression then
Statement
end if
```

但是在 Java 中给出了这样的定义：

```
IfThenStatement:
   if ( Expression ) Statement
```

即在一个 if 语句中，表示条件的表达式必须用小括号标示，同时在右小括号后，书写语句块，表示执行内容。而对于 Expression 和 Statement 的具体定义，在语言规范中也有十分详细的描述，这里就不一一展开了，有兴趣的读者可以参考 Java 语言规范，JDK 1.7 版第 14 章的内容“Blocks and Statements”。

1.3.3 数据类型的定义

Java 语言规范中还定义了 Java 的数据类型。根据 Java 1.7 的规范，Java 的数据类型分为原始数据类型和引用数据类型。原始数据类型又分为数字型和布尔型。数字型又有 byte、short、int、long、char、float、double。注意，在这里 char 被定义为整数型，并且在规范中明确定义：byte、short、int、long 分别是 8 位、16 位、32 位、64 位有符号整数，而 char 为 16 位无符号整数，表示 UTF-16 的字符。布尔型只有两种取值：true 和 false。而对于 float 和 double，规范中规定，它们是满足 IEEE 754 的 32 位浮点数和 64 位浮点数。

注意： 在 Java 语言中，char 占 2 字节，而不是 C 语言中的 1 字节。从这点上看，Java 的国际化是在语言底层就提供了强有力的支持。

此外，规范还定义了各类数字的取值范围、初始值，以及能够支持的各种操作。以整数为例，比较运算、数值运算、位运算、自增自减运算等都在规范中有描述。

除了基本数据类型外，引用数据类型也是 Java 重要的组成部分，引用数据类型分为 3 种：类或接口、泛型类型以及数组类型。

提醒： 引用类型和原始类型在 Java 的处理中是截然不同的，尤其对于它们的“相等”操作。

【示例 1-1】在 Java 语言规范中，有一个简短的示例，说明了引用类型和原始类型的区别：

```
class Value { int val; }
class Test {
   public static void main(String[] args) {
       int i1 = 3;
       int i2 = i1;
       i2 = 4;
       System.out.print("i1==" + i1);
       System.out.println(" but i2==" + i2);
       Value v1 = new Value();
       v1.val = 5;
       Value v2 = v1;
```

```
        v2.val = 6;
        System.out.print("v1.val==" + v1.val);
        System.out.println(" and v2.val==" + v2.val);
    }
}
```

上述程序将输出：

```
i1==3 but i2==4
v1.val==6 and v2.val==6
```

从上述输出可以看出，对于原始数据类型 int，i1 和 i2 表示不同的变量，两者毫无关系，但是对于 v1 和 v2，它们都指向唯一一个由 new 关键字创建的 Value 对象。

由于本书并非讲解 Java 语言，因此对于这部分内容点到即止，有兴趣的读者可以参考 Java 语言规范的第 4 章“Types, Values, and Variables”。

1.3.4　Java 语言规范总结

除上述基本内容外，Java 语言规范还定义了各种不同类型间的转换规则、方法的可见性定义、有关接口的使用、注释等。

总之，Java 语言规范完整定义和描述了 Java 语言的所有特性，因为 Java 语言本身不属于本书的讨论重点，故在此只做简要介绍。

1.4　一切听我的：Java 虚拟机规范

虽然 Java 语言和 Java 虚拟机有着密切的联系，但两者是完全不同的内容。Java 虚拟机是一台执行 Java 字节码的虚拟计算机，它拥有独立的运行机制，其运行的 Java 字节码也未必由 Java 语言编译而成，像 Groovy、Scala 等语言生成的 Java 字节码也可以由 Java 虚拟机执行。立足于 Java 虚拟机，可以产生各种各样的跨平台语言。除了语言特性各不相同外，它们可以共享 Java 虚拟机带来的跨平台性、优秀的垃圾回收器，以及可靠的即时编译器。

因此，与 Java 语言不同，Java 虚拟机是一个高效的、性能优异的、商用级别的软件运行和开发平台，而这也是本书讨论的重点。

Java 虚拟机规范的主要内容大概有以下几个部分：

- 定义了虚拟机的内部结构（将在第 2 章中详细介绍）。

- 定义了虚拟机执行的字节码类型和功能（将在第 11 章中详细介绍）。
- 定义了 Class 文件的结构（将在第 9 章中详细介绍）。
- 定义了类的装载、连接和初始化（将在第 10 章中详细介绍）。

以 Java 1.7 为例，读者可以在 http://docs.oracle.com/javase/specs/jvms/se7/html/浏览虚拟机规范全文。这份规范可以说是开发 Java 虚拟机的指导性文件，如果要实现自定义的 Java 虚拟机，则需要参考和熟悉这份规范，同时这份规范对于了解现存的流行 Java 虚拟机（如 Hotspot、IBM J9 等），也有十分重要的意义。

1.5 数字编码就是计算机世界的水和电

数字是计算机内最直接、最基础的表现类型。了解数字在计算机内的表示，对于了解整个计算机系统具有相当重要的作用，数字也是专业计算机从业人员的基本功之一。本节将主要介绍整数以及浮点数在 Java 虚拟机中的支持情况。

1.5.1 整数在 Java 虚拟机中的表示

在 Java 虚拟机中，整数有 byte、short、int、long 四种，分别表示 8 位、16 位、32 位、64 位有符号整数。整数在计算机中使用补码表示，在 Java 虚拟机中也不例外。在学习补码之前，必须先理解原码和反码。

所谓原码，就是符号位加上数字的二进制表示。以 int 为例，第 1 位表示符号位（正数或者负数），其余 31 位表示该数字的二进制值。

10 的原码为：00000000 00000000 00000000 00001010

-10 的原码为：10000000 00000000 00000000 00001010

对于原码来说，绝对值相同的正数和负数只有符号位不同。

反码就是在原码的基础上，符号位不变，其余位取反，以-10 为例，其反码为：

11111111 11111111 11111111 11110101

负数的补码就是反码加 1，整数的补码就是原码本身。

因此，10 的补码为：

00000000 00000000 00000000 00001010

而-10 的补码为：

11111111 11111111 11111111 11110110

在 Java 中，可以使用位运算查看整数中每一位的实际值，方法如下：

```
01  int a=-10;
02  for(int i=0;i<32;i++){
03      int t=(a & 0x80000000>>>i)>>>(31-i);
04      System.out.print(t);
05  }
```

以上代码将打印-10 在虚拟机内的实际表示，程序的执行结果如下：

```
11111111111111111111111111110110
```

可以看到，这个结果和之前补码的计算结果是完全匹配的。

这段程序的基本思想是：进行 32 次循环（因为 int 有 32 位），每次循环取出 int 值中的一位，第 3 行的 0x80000000 是一个首位为 1、其余位为 0 的整数，通过右移 i 位，定位到要获取的第 i 位，并将除该位外的其他位统一设置为 0，而该位不变，最后将该位移至最右，并进行输出。

相对于原码，使用补码作为计算机内的实际存储方式至少有以下两个好处。

（1）可以统一数字 0 的表示。由于 0 既非正数，又非负数，使用原码表示时符号位难以确定，把 0 归入正数或者负数得到的原码编码是不同的。但是使用补码表示时，无论把 0 归入正数或者负数都会得到相同的结果。计算过程如下：

如果 0 为正数，则补码为原码本身为：00000000 00000000 00000000 00000000。

如果 0 为负数，则补码为反码加 1，负数 0 的原码为：

10000000 00000000 00000000 00000000

反码为：

11111111 11111111 11111111 11111111

补码在反码的基础上加 1，结果为：

00000000 00000000 00000000 00000000

可以看到，使用补码作为整数编码，可以解决数字 0 的存储问题。

（2）使用补码可以简化整数的加减法计算，将减法计算视为加法计算，实现减法和加法的完全统一，实现正数和负数加法的统一。现使用 8 位（byte）整数说明这个问题。

计算-6+5 的过程如下。

-6 补码：11111010

5 补码：00000101

直接相加得：11111111

通过计算可知，11111111 表示-1。

计算 4+6 的过程如下。

4 的补码：00000100

6 的补码：00000110

直接相加得：00001010

通过计算可知，00001010 表示 10（十进制）。

可以看到，使用补码表示时，只需要将补码简单地相加，即可得到算术加法的正确结果，而无须区别正数或者负数。

1.5.2 浮点数在 Java 虚拟机中的表示

在 Java 虚拟机中，浮点数有 float 和 double 两种，分别是 32 位和 64 位浮点数。浮点数在虚拟机中的表示比整数略显复杂。目前，使用最为广泛的是由 IEEE 754 定义的浮点数格式。目前 Java 虚拟机中对于浮点数的处理参考 IEEE 754 的规范。本节将以 float 为例，简要介绍浮点数的表示方法。

在 IEEE 754 的定义中，一个浮点数由 3 部分组成，分别是：符号位、指数位和尾数位。以 32 位 float 为例，符号位占 1 位，表示正负数，指数位占 8 位，尾数位占剩余的 23 位，如图 1.2 所示。

其中，sflag 表示符号，当 s 为 0 时，sflag 为 1，当 s 为 1 时，sflag 为-1。m 为尾数值，实际占用空间为 23 位，但是根据 e 的取值，有 24 位精度。当 e 全为 0 时，尾数位附加为 0，否则，尾数位附加为 1。e 为指数位，用 8 位表示。

图 1.2　浮点数格式

以浮点数-5 为例，其内部表示为：

1 10000001 01000000000000000000000

符号位为 1 表示负数，指数位为 10000001，表示 129。

尾数位为：01000000000000000000000。因为 e 不全为 0，故实际的尾数位为：

101000000000000000000000

尾数位表示 2 的指数次幂的和，每一位表示求和数列中的对应项是否为 0，这里表示：

$1*2^0+0*2^{-1}+1*2^{-2}+0*2^{-3}+0*2^{-4}+0*2^{-5}$....，对应关系如图 1.3 所示。

图 1.3　尾数的计算

故 1 10000001 01000000000000000000000 的值为：

$-1*2^{(129-127)}*(1*2^0+0*2^{-1}+1*2^{-2}+0*2^{-3}+0*2^{-4}+0*2^{-5}) = -1*4*1.25 = -5$

浮点数 float 还可以表示一些特殊的数字，如表 1.1 所示。

表 1.1　float的特殊数字

正无穷	0 11111111 00000000000000000000000
负无穷	1 11111111 00000000000000000000000
NaN	0 11111111 10000000000000000000000
最大浮点数(3.4028235E38)	0 11111110 11111111111111111111111
最小规范化正浮点数(1.17549435E-38)	0 00000001 00000000000000000000000
最小正浮点数(1.4E-45)	0 00000000 00000000000000000000001
0	0 00000000 00000000000000000000000

其中，指数位全为 1 的表示无穷大和 NaN 等特殊数字。指数位全为 0 的为非规范化的浮点数。

【示例 1-2】在 Java 中，使用 Float. floatToRawIntBits()函数可以获得一个单精度浮点数的 IEEE 754 表示。以下代码打印了-5 的内部表示：

```
float a=-5;
System.out.println(Integer.toBinaryString(Float.floatToRawIntBits(a)));
```

程序运行结果为：

```
11000000101000000000000000000000
```

其中，Float.floatToRawIntBits()函数最终由 native 方法实现，具体实现如下：

```
JNIEXPORT jint JNICALL
Java_java_lang_Float_floatToRawIntBits(JNIEnv *env, jclass unused, jfloat v)
{
    union {
        int i;
        float f;
    } u;
    u.f = (float)v;
    return (jint)u.i;
}
```

从上述代码可以看出，为了获取 float 的内部表示，使用了 C 语言中的 union 自然实现这个转换。

1.6　抛砖引玉：编译和调试虚拟机

如果要对虚拟机进行深入的研究，那么编译和调试 Java 虚拟机是必不可少的。为何要编译

自己的虚拟机呢？主要原因有两点。

第一，通过自己编译可以得到一个 debug 或者 fastdebug 版本的调试用虚拟机，调试用虚拟机可以支持更多的虚拟机参数，这些开发专用的虚拟机参数可以帮助开发人员获得更多的虚拟机内部信息，而这些参数在正常发行版本中是无法使用的。因此考虑到本书的实用性，本书并不会过多介绍那些只在调试版本中使用的参数，但如果读者有兴趣，可以编译自己的虚拟机，进行尝试。

第二，使用自己编译的调试版的虚拟机可以用于虚拟机代码的单步调试，有利于实现对虚拟机代码的理解。由于虚拟机代码比较复杂，仅通过代码阅读很难深入理解其实现机制，有时不得不依靠单步调试进行辅助，而编译后调试版本可以帮助实现这一功能。

为编译虚拟机，首先必须获得虚拟机源码，读者可以在 OpenJDK 的官方网站上下载最新源码，下载地址为 https://jdk7.java.net/source.html。笔者目前使用的是 openjdk-7u40-fcs-src-b43-26_aug_2013.zip。推荐读者使用较新的版本，因为老版本的编译脚本可能在某些平台上存在问题。

笔者的编译环境为 Ubuntu 系统，读者可以选择自己喜欢的 Linux 发行版进行编译。编译虚拟机之前，还必须做一些准备工作。

1. 编译前的准备工作

（1）安装好依赖库。在 Ubuntu 平台上可以通过 apt-get 命令安装，在 CentOS 平台上可以通过 yum 命令安装。比如，笔者在编译前至少安装了以下库：

```
apt-get install build-essential
apt-get install libxrender-devsudo
apt-get install xorg-devsudo
apt-get install libasound2-devsudo
apt-get install libcups2-dev
apt-get install libasound2-dev
apt-get install libfreetype6-dev
apt-get install libcups2-dev
apt-get install zip
apt-get install libX11-dev
apt-get install libxext-dev
apt-get install libxrender-dev
apt-get install libxtst-dev
apt-get install libxt-dev
```

如果依赖库安装不全，在编译过程中就会提示错误，从错误提示中，读者应该可以得知缺少了哪些库或者头文件，相应地安装即可。当然了，gcc 和 g++作为基本的编译工具也是必须要

安装的。

（2）准备一个 Boot JDK 和 ANT。Boot JDK 用于 OpenJDK 编译的执行。笔者在这里使用的是 jdk1.6.0u45，也推荐读者使用 JDK 1.6 的版本作为 JDK 1.7 的 Boot JDK。此外，还需要准备 ANT 1.7.1 以上版本，作为编译的基础执行工具。

2. 准备编译

准备就绪之后，就可以开始准备编译了。

（1）进入解压后的 openjdk 目录：

```
geym@hzlab001:~/openjdk/openjdk$ ls
ASSEMBLY_EXCEPTION build.sh get_source.sh hotspot.log jaxws langtools
make README test
build corba hotspot jaxp jdk LICENSE Makefile README-builds.html THIRD_
PARTY_README
```

（2）新建文件 build.sh，作为编译启动执行脚本。建立此脚本的目的是方便编译执行，因为编译前可能会预设一些环境变量，写成脚本后更加容易维护和执行。读者可以根据自己的执行环境，自行修改，笔者的脚本如下：

```
#!/bin/bash
export ANT_HOME=/home/geym/tools/apache-ant-1.9.4
export PATH=$ANT_HOME/bin:$PATH
export ALT_BOOTDIR=/home/geym/tools/jdk1.6.0_45
unset JAVA_HOME
export ALT_JDK_IMPORT_PATH=/home/geym/tools/jdk1.6.0_45
unset CLASSPATH

export SKIP_DEBUG_BUILD=false
export SKIP_FASTDEBUG_BUILD=false
export DEBUG_NAME=debug
export LANG=C

cd /geym/home/openjdk/openjdk
make sanity && make BUILD_JAXWS=false BUILD_JAXP=false  WARNINGS_ARE_ERRORS=
```

该脚本中，首先设置了 ANT_HOME，并将 ANT 执行目录加入 PATH，接着设置了 Boot JDK 的路径。这里注意，需要 unset JAVA_HOME 以及 CLASSPATH 变量。

（3）通过 make 命令执行整个编译，根据这个 build 脚本，将会同时生成 debug 版本、

fastdebug 和常规发行版本三个虚拟机的编译结果。编译的过程可能会花费比较长的时间，一般来说，编译一个版本可能需要 15 到 45 分钟，视计算机性能而定。当编译成功后，会有以下输出，显示了编译耗时。

```
#-- Build times ----------
Target debug_build
Start 2014-11-10 18:04:19
End   2014-11-10 18:21:14
00:01:27 corba
00:07:18 hotspot
00:07:43 jdk
00:00:26 langtools
00:16:55 TOTAL
-------------------------
```

进入 build 目录，可以看到编译的结果，下面显示了 3 个版本的编译结果：

```
geym@:~/openjdk/openjdk/build$ du -h --max-depth=1
2.5G    ./linux-amd64-fastdebug
2.4G    ./linux-amd64-debug
2.4G    ./linux-amd64
```

有了 debug 版本的虚拟机，就可以使用 gdb 对虚拟机进行调试了。接下来将简单地演示 Java 虚拟机的调试方法。

3. Java 虚拟机的调试

笔者选用 linux-amd64-debug 下的虚拟机进行调试。

（1）首先进入 linux-amd64-debug 目录，查找名为 gamma 的文件。

```
geym@:~/openjdk/openjdk/build/linux-amd64-debug$ find -name gamma
./hotspot/outputdir/linux_amd64_compiler2/jvmg/gamma
```

该文件就是目标文件，用于调试，实际上，它就是 java 程序的 debug 版本。这里记下文件的路径。

（2）接着，进入含有 Java Class 执行文件的目录下并启动 gdb。启动 gdb 后，先为 gamma 程序的运行设定环境变量：

```
(gdb) set environment JAVA_HOME=/home/geym/tools/jdk1.6.0_45
(gdb) set environment  CLASSPATH=.:${JAVA_HOME}/jre/lib/rt.jar:${JAVA_HOME}/
jre/lib/i18n.jar
(gdb) set environment  HOTSPOT_BUILD_USER="geym in hotspot"
```

```
    (gdb) set environment LD_LIBRARY_PATH=/home/geym/openjdk/openjdk/build/linux-amd64-debug/hotspot/outputdir/linux_amd64_compiler2/jvmg:$JAVA_HOME/jre/lib/amd64:$JAVA_HOME/jre/lib/amd64/native_threads
```

（3）使用 file 指令将 gamma 程序设为目标文件，此时应该会显示如下信息：

```
    (gdb) file /home/geym/openjdk/openjdk/build/linux-amd64-debug/hotspot/outputdir/linux_amd64_compiler2/jvmg/gamma
    Reading symbols from /home/geym/openjdk/openjdk/build/linux-amd64-debug/hotspot/outputdir/linux_amd64_compiler2/jvmg/gamma...done.
```

这表示，读取调试符号信息成功。

（4）设置传递给 gamma 程序的参数，也就是传递给 Java 程序的参数，一般为需要执行的 Main Class 的名称，这里运行 SimpleGc 类，并为 SimpleGc 程序传入参数 ggg，这相当于执行 java SimpleGc ggg。

```
    (gdb) set args SimpleGc ggg
```

（5）设置断点，这里在 Java 进程的 main()函数中设置断点。

```
    (gdb) break main
    Breakpoint 1 at 0x403a02: file /home/geym/openjdk/openjdk/hotspot/src/ share/tools/launcher/java.c, line 228.
```

（6）执行 run 命令启动 Java 程序，开始调试。

```
    (gdb) run
    Starting program: /home/geym/openjdk/openjdk/build/linux-amd64-debug/hotspot/outputdir/linux_amd64_compiler2/jvmg/gamma SimpleGc ggg
    [Thread debugging using libthread_db enabled]
    Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

    Breakpoint 1, main (argc=3, argv=0x7fffffffe3d8) at /home/geym/openjdk/openjdk/hotspot/src/share/tools/launcher/java.c:228
    228      {
    (gdb) n
    省略部分输出
    (gdb)
    329      if (!ParseArguments(&argc, &argv, &jarfile, &classname, &ret, jvmpath)) {
    (gdb) p argv
    $4 = (char **) 0x7fffffffe3e0
    (gdb) p *argv
    $7 = 0x7fffffffe695 "SimpleGc"
```

```
(gdb) p *(argv+1)
$8 = 0x7fffffffe69e "ggg"
```

可以看到，当前 Java 程序从 java.c 文件的第 228 行开始执行。使用命令 next（缩写 n），进行单步调试，这里省略了中间部分输出，在运行到 ParseArguments()函数时，通过 print（缩写 p）命令显示了传递给 Java 进程的参数，这里可以看到作为 Main Class 的 SimpleGc 及其参数“ggg”。

至此，读者就可以使用这套环境作为辅助，更好地深入 Java 虚拟机内部了。

1.7 小结

本章主要介绍了 Java 语言和 Java 虚拟机的发展历史，并介绍了 Java 生态环境中两份非常重要的规范——Java 语言规范和 Java 虚拟机规范。其中 Java 虚拟机规范将成为本书后续讨论的重点内容。同时，作为了解 Java 虚拟机的第一步，简要介绍了整数和浮点数在 Java 虚拟机中的表示方式。最后，为了方便读者更加深入地了解虚拟机，还给出了单步调试 Java 虚拟机的方法。

2

第 2 章 认识Java虚拟机的基本结构

Java 虚拟机那么复杂，它的基本结构是什么？各个组成部分有何作用？又是如何相互协调工作的呢？本章将试图解答这些问题，而要解答这些问题就必须先了解 Java 堆、Java 栈、永久区和元数据区的基本概念。

本章涉及的主要知识点有：

- 认识 Java 虚拟机中的堆。
- 了解有关栈的概念和使用。
- 了解存放类型描述的永久区和元数据区。

2.1 谋全局者才能成大器：看穿 Java 虚拟机的架构

Java 虚拟机的基本结构如图 2.1 所示。

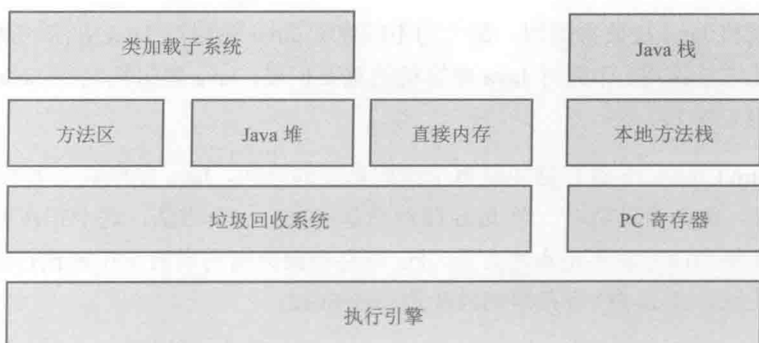


图 2.1　Java 虚拟机的基本结构

类加载子系统负责从文件系统或者网络中加载 Class 信息，加载的类信息存放于一块称为方法区的内存空间。除了类的信息外，方法区中可能还会存放运行时常量池信息，包括字符串字面量和数字常量（这部分常量信息是 Class 文件中常量池部分的内存映射）。

Java 堆在虚拟机启动的时候建立，它是 Java 程序最主要的内存工作区域。几乎所有的 Java 对象实例都存放于 Java 堆中。堆空间是所有线程共享的，这是一块与 Java 应用密切相关的内存区间。

Java 的 NIO 库允许 Java 程序使用直接内存。直接内存是在 Java 堆外的、直接向系统申请的内存区间。通常，访问直接内存的速度会优于 Java 堆。因此出于性能考虑，读写频繁的场合可能会考虑使用直接内存。由于直接内存在 Java 堆外，因此它的大小不会直接受限于 Xmx 指定的最大堆大小，但是系统内存是有限的，Java 堆和直接内存的总和依然受限于操作系统能给出的最大内存。

垃圾回收系统是 Java 虚拟机的重要组成部分，垃圾回收器可以对方法区、Java 堆和直接内存进行回收。其中，Java 堆是垃圾收集器的工作重点。和 C/C++不同，Java 中所有的对象空间释放都是隐式的。也就是说，Java 中没有类似 free()或者 delete()这样的函数释放指定的内存区域。对于不再使用的垃圾对象，垃圾回收系统会在后台默默工作，默默查找、标识并释放垃圾对象，完成包括 Java 堆、方法区和直接内存中的全自动化管理。有关垃圾回收系统的更多信息，可以参阅第 4 章和第 5 章。

每一个 Java 虚拟机线程都有一个私有的 Java 栈。一个线程的 Java 栈在线程创建的时候被创建。Java 栈中保存着帧信息（参阅本章 2.4 节），Java 栈中保存着局部变量、方法参数，同时和 Java 方法的调用、返回密切相关。

本地方法栈和 Java 栈非常类似，最大的不同在于 Java 栈用于 Java 方法的调用，而本地方法栈则用于本地方法调用。作为对 Java 虚拟机的重要扩展，Java 虚拟机允许 Java 直接调用本地方法（通常使用 C 编写）。

PC（Program Counter）寄存器也是每个线程私有的空间，Java 虚拟机会为每一个 Java 线程创建 PC 寄存器。在任意时刻，一个 Java 线程总是在执行一个方法，这个正在被执行的方法称为当前方法。如果当前方法不是本地方法，PC 寄存器就会指向当前正在被执行的指令。如果当前方法是本地方法，那么 PC 寄存器的值就是 undefined。

执行引擎是 Java 虚拟机的最核心组件之一，它负责执行虚拟机的字节码。现代虚拟机为了提高执行效率，会使用即时编译技术将方法编译成机器码后再执行。执行引擎的进一步细节描述可以参阅第 11 章。

2.2 小参数能解决大问题：学会设置 Java 虚拟机的参数

Java 虚拟机可以使用 JAVA_HOME/bin/java 程序启动（JAVA_HOME 为 JDK 的安装目录），一般来说，Java 进程的命令行使用方法如下：

```
java [-options] class [args...]
```

其中，-options 表示 Java 虚拟机的启动参数，class 为带有 main()函数的 Java 类，args 表示传递给主函数 main()的参数。

如果需要设定特定的 Java 虚拟机参数，在 options 处指定即可。目前，Hotspot 虚拟机支持大量的虚拟机参数，可以帮助开发人员进行系统调优和故障排查。相关的一些参数将在本书的后续章节中逐步展开介绍，本节则主要介绍参数的设置方法。

【示例 2-1】以如下代码为例，读者先来了解一下如何设置参数。

```
package geym.zbase.ch2;

public class SimpleArgs {
   public static void main(String[] args) {
       for(int i=0;i<args.length;i++){
           System.out.println("参数"+(i+1)+":"+args[i]);
       }
       System.out.println("-Xmx"+Runtime.getRuntime().maxMemory()/1000/1000+"M");
   }
}
```

上述代码打印了传递给 main()函数的参数，同时打印了系统的最大可用堆内存。使用如下命令行运行这段代码：

```
java  -Xmx32m geym.zbase.ch2.SimpleArgs a
参数 1:a
-Xmx32M
```

从结果可以看到，第一个参数-Xmx32m 传递给 Java 虚拟机，生效后，使得系统最大可用堆空间为 32MB，参数 a 则传递给主函数 main()，作为应用程序的参数。

有关-Xmx 会在本书后续章节中展开讨论，除了-Xmx 外，虚拟机还支持大量的调优诊断参数，其设置方式都是类似的，在本书后续章节中会逐步介绍这些参数。

如果读者使用 Eclipse 等开发工具运行程序，在运行对话框的参数选项卡上，也可以设置这两个参数，如图 2.2 所示，显示了“程序参数”和“虚拟机参数”两个文本框，将所需的参数填入即可。



图 2.2 通过 Eclipse 为虚拟机设置启动参数

2.3 对象去哪儿：辨清 Java 堆

Java 堆是和 Java 应用程序关系最为密切的内存空间，几乎所有的对象都存放在堆中。并且 Java 堆是完全自动化管理的，通过垃圾回收机制，垃圾对象会被自动清理，而不需要显式地释放。

根据垃圾回收机制的不同，Java 堆有可能拥有不同的结构。最为常见的一种构成是将整个 Java 堆分为新生代和老年代。其中，新生代存放新生对象或者年龄不大的对象，老年代则存放老年对象。新生代有可能分为 eden 区、s0 区、s1 区，s0 和 s1 也被称为 from 和 to 区域，它们是两块大小相等、可以互换角色的内存空间。详细信息可以参阅第 4 章。

图 2.3 显示了一个堆空间的一般结构。

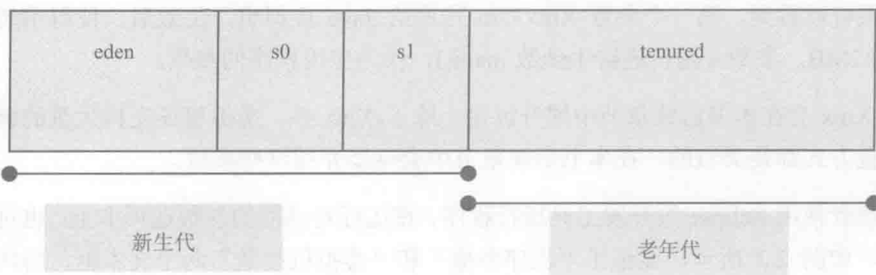


图 2.3 堆空间的一般结构

在绝大多数情况下，对象首先分配在 eden 区，在一次新生代回收后，如果对象还存活，则会进入 s0 或者 s1，之后，每经过一次新生代回收，对象如果存活，它的年龄就会加 1。当对象的年龄达到一定条件后，就会被认为是老年对象，从而进入老年代。

【示例 2-2】下面通过一个简单的示例，来展示 Java 堆、方法区和 Java 栈之间的关系。

```
public class SimpleHeap {
    private int id;
    public SimpleHeap(int id){
        this.id=id;
    }
    public void show(){
        System.out.println("My ID is "+id);
    }
    public static void main(String[] args) {
        SimpleHeap s1=new SimpleHeap(1);
        SimpleHeap s2=new SimpleHeap(2);
        s1.show();
        s2.show();
    }
}
```

上述代码声明了一个 SimpleHeap 类，并在 main()函数中创建了两个 SimpleHeap 实例。此时，各对象和局部变量的存放如图 2.4 所示。SimpleHeap 实例本身分配在堆中，描述 SimpleHeap

类的信息存放在方法区，main()函数中 s1 和 s2 局部变量存放在 Java 栈中，并指向堆中的两个实例。

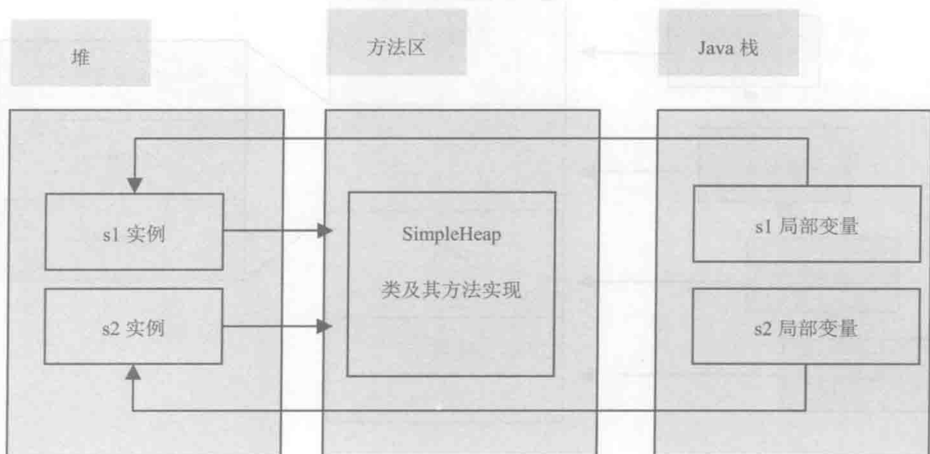


图 2.4　堆、方法区、栈的关系

2.4　函数如何调用：出入 Java 栈

Java 栈是一块线程私有的内存空间。如果说，Java 堆和程序数据密切相关，那么 Java 栈就是和线程执行密切相关的。线程执行的基本行为是函数调用，每次函数调用的数据都是通过 Java 栈传递的。

Java 栈与数据结构上的栈有着类似的含义，它是一块先进后出的数据结构，只支持出栈和入栈两种操作。在 Java 栈中保存的主要内容为栈帧。每一次函数调用，都会有一个对应的栈帧被压入 Java 栈，每一个函数调用结束，都会有一个栈帧被弹出 Java 栈。如图 2.5 所示，函数 1 对应栈帧 1，函数 2 对应栈帧 2，依此类推。函数 1 中调用函数 2，函数 2 中调用函数 3，函数 3 中调用函数 4。当函数 1 被调用时，栈帧 1 入栈；当函数 2 被调用时，栈帧 2 入栈；当函数 3 被调用时，栈帧 3 入栈；当函数 4 被调用时，栈帧 4 入栈。当前正在执行的函数所对应的帧就是当前的帧（位于栈顶），它保存着当前函数的局部变量、中间运算结果等数据。

当函数返回时，栈帧从 Java 栈中被弹出。Java 方法有两种返回函数的方式，一种是正常的函数返回，使用 return 指令；另外一种是抛出异常。不管使用哪种方式，都会导致栈帧被弹出。

在一个栈帧中，至少要包含局部变量表、操作数栈和帧数据区几个部分。

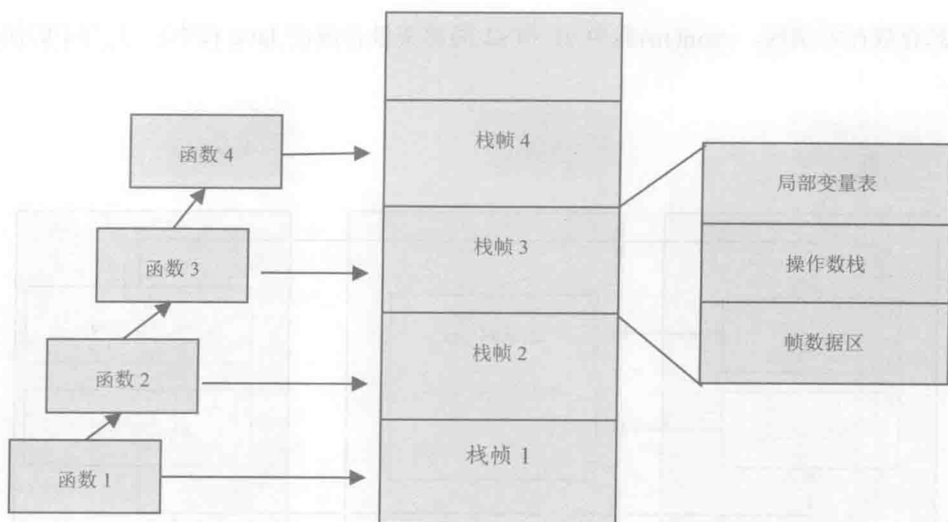


图 2.5 帧栈和函数调用

提示： 由于每次函数调用都会生成对应的栈帧，从而占用一定的栈空间，因此，如果栈空间不足，那么函数调用自然无法继续进行下去。当请求的栈深度大于最大可用栈深度时，系统就会抛出 StackOverflowError 栈溢出错误。

Java 虚拟机提供了参数-Xss 来指定线程的最大栈空间，这个参数也直接决定了函数调用的最大深度。

【示例 2-3】下面的代码是一个递归调用，由于递归没有出口，这段代码可能会出现栈溢出错误，在抛出错误后，程序打印了最大的调用深度。

```
public class TestStackDeep {
    private static int count=0;
    public static void recursion(){
        count++;
        recursion();
    }
    public static void main(String args[]){
        try{
            recursion();
        }catch(Throwable e){
            System.out.println("deep of calling = "+count);
            e.printStackTrace();
        }
```

```
    }
}
```

使用参数-Xss128K 执行以上代码，部分结果如下所示：

```
deep of calling = 2505
java.lang.StackOverflowError
   at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:16)
   at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:17)
   ….
```

可以看到，在进行大约 2500 次调用后，发生了栈溢出错误，通过增大-Xss 的值，可以获得更深的调用层次，尝试使用参数-Xss256K 执行上述代码，可能产生如下输出，很明显，调用层次有明显的增加。

```
deep of calling = 5809
java.lang.StackOverflowError
   at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:16)
   at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:17)
```

注意： 函数嵌套调用的层次在很大程度上由栈的大小决定，栈越大，函数可以支持的嵌套调用次数就越多。

2.4.1 局部变量表

局部变量表是栈帧的重要组成部分之一。它用于保存函数的参数以及局部变量。局部变量表中的变量只在当前函数调用中有效，当函数调用结束后，随着函数栈帧的销毁，局部变量表也会随之销毁。

由于局部变量表在栈帧之中，因此，如果函数的参数和局部变量较多，会使得局部变量表膨胀，从而每一次函数调用就会占用更多的栈空间，最终导致函数的嵌套调用次数减少。

【示例 2-4】下面的代码演示了这种情况，第 1 个 recursion()函数含有 3 个参数和 10 个局部变量，因此，其局部变量表含有 13 个变量。而第 2 个 recursion()函数不含有任何参数和局部变量。当这两个函数被嵌套调用时，第 2 个 recursion()函数可以拥有更深的调用层次。

```
public class TestStackDeep {
   private static int count=0;
   public static void recursion(long a,long b,long c){
      long e=1,f=2,g=3,h=4,i=5,k=6,q=7,x=8,y=9,z=10;
      count++;
      recursion(a,b,c);
```

```
    }
    public static void recursion(){
        count++;
        recursion();
    }
    public static void main(String args[]){
        try{
            // recursion(0L,0L,0L);
            recursion();
        }catch(Throwable e){
            System.out.println("deep of calling = "+count);
            e.printStackTrace();
        }
    }
}
```

使用参数-Xss128K 执行上述代码中的第 1 个 recursion()函数，输出结果如下：

```
deep of calling = 692
java.lang.StackOverflowError
    at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:11)
```

使用参数-Xss128K 执行上述代码中的第 2 个 recursion()函数，输出结果如下：

```
deep of calling = 2496
java.lang.StackOverflowError
    at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:16)
```

可以看到，在相同的栈容量下，局部变量少的函数可以支持更深的函数调用。

使用 jclasslib 工具可以更进一步查看函数的局部变量信息。图 2.6 显示了第一个 recursion()函数的最大局部变量表的大小为 26 个字。因为该函数包含总共 13 个参数和局部变量，且都为 long 型，long 和 double 在局部变量表中需要占用 2 个字，其他如 int、short、byte、对象引用等占用 1 个字。

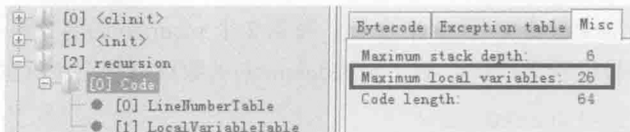


图 2.6 最大局部变量表大小

说明： 字（Word）指的是计算机内存中占据一个单独的内存单元编号的一组二进制串。一般 32 位计算机上一个字为 4 个字节长度。

图 2.7 显示了在 Class 文件中的局部变量表的内容（这里说的局部变量表和上述的局部变量表不同，这里指 Class 文件的一个属性，而上述的局部变量表指 Java 栈空间的一部分）。

Methods
[0] <clinit>
[1] <init>
[2] recursion
[0] Code
[0] LineNumberTable
[1] LocalVariableTable
[3] recursion
[0] Code
[0] LineNumberTable
[1] LocalVarisbleTable
[4] main
Attributes

Specific info:

Nr.	start_pc	length	index	name	descriptor
0	0	64	0	cp info #41 a	cp info #42 J
1	0	64	2	cp info #43 b	cp info #42 J
2	0	64	4	cp info #44 c	cp info #42 J
3	3	61	6	cp info #45 e	cp info #42 J
4	8	56	8	cp info #46 f	cp info #42 J
5	13	51	10	cp info #47 g	cp info #42 J
6	18	46	12	cp info #48 h	cp info #42 J
7	23	41	14	cp info #49 i	cp info #42 J
8	28	36	16	cp info #50 k	cp info #42 J
9	33	31	18	cp info #51 q	cp info #42 J
10	38	26	20	cp info #52 x	cp info #42 J
11	43	21	22	cp info #53 y	cp info #42 J
12	48	16	24	cp info #54 z	cp info #42 J

图 2.7　Class 文件中的局部变量表

可以看到，在 Class 文件的局部变量表中，显示了每个局部变量的作用域范围、所在槽位的索引（index 列）、变量名（name 列）和数据类型（J 表示 long 型）。

栈帧中的局部变量表中的槽位是可以重用的，如果一个局部变量过了其作用域，那么在其作用域之后申明的新的局部变量就很有可能会复用过期局部变量的槽位，从而达到节省资源的目的。

【示例 2-5】下面的代码显示了局部变量表槽位的复用。在 localvar1()函数中，局部变量 a 和 b 都作用到了函数末尾，故 b 无法复用 a 所在的位置。而在 localvar2()函数中，局部变量 a 在第 10 行时不再有效，故局部变量 b 可以复用 a 的槽位（1 个字）。

```
01 public void localvar1(){
02     int a=0;
03     System.out.println(a);
04     int b=0;
05 }
06 public void localvar2(){
07     {
08     int a=0;
```

```
    System.out.println(a);
    }
    int b=0;
}
```

图 2.8 显示了 localvar1()的局部变量信息，该函数最大局部变量大小为 3 字，第 0 个槽位为函数的 this 引用（实例方法的第一个局部变量都是 this 引用），第 1 个槽位为变量 a，第 2 个槽位为变量 b，每个变量占 1 字，合计 3 字。

Maximum stack depth:	1
Maximum local variables:	3
Code length:	5

Nr.	start_pc	length	index	name	descriptor
0	0	5	0	cp info #12 this	cp info #13 Lgeym/zbase/ch2/localvar/Loca...
1	2	3	1	cp info #15 a	cp info #16 I
2	4	1	2	cp info #17 b	cp info #16 I

图 2.8　localvar1()的局部变量信息

图 2.9 显示了 localvar2()的局部变量信息，该函数的最大局部变量表为 2 字，虽然和 localvar1()一样，拥有 this、a、b 等 3 个局部变量，但 b 复用了 a 的槽位（从它们都占用了第 1 个槽位可以知道这点），因此在整个函数执行中，同时存在的最大局部变量为 2 字。

Maximum stack depth:	2
Maximum local variables:	2
Code length:	12

Nr.	start_pc	length	index	name	descriptor
0	0	12	0	cp info #12 this	cp info #13 Lgeym/zbase/ch2/localvar/Loca...
1	2	7	1	cp info #15 a	cp info #16 I
2	11	1	1	cp info #17 b	cp info #16 I

图 2.9　localvar2()的局部变量信息

局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引用的对象都是不会被回收的。因此，理解局部变量表对理解垃圾回收也有一定帮助。

【示例 2-6】下面通过一个简单的示例，展示局部变量对垃圾回收的影响。

```
public void localvarGc1(){
    byte[] a=new byte[6*1024*1024];
```

```
        System.gc();
    }
    public void localvarGc2(){
        byte[] a=new byte[6*1024*1024];
        a=null;
        System.gc();
    }
    public void localvarGc3(){
        {
        byte[] a=new byte[6*1024*1024];
        }
        System.gc();
    }
    public void localvarGc4(){
        {
        byte[] a=new byte[6*1024*1024];
        }
        int c=10;
        System.gc();
    }
    public void localvarGc5(){
        localvarGc1();
        System.gc();
    }
    public static void main(String[] args) {
        LocalVarGC ins=new LocalVarGC();
        ins.localvarGc1();
    }
```

上述代码中，每一个 localvarGc*N*()函数都分配了一块 6MB 的堆空间，并使用局部变量引用这块空间。

在 localvarGc1()中，在申请空间后，立即进行垃圾回收，很明显，由于 byte 数组被变量 a 引用，因此无法回收这块空间。

在 localvarGc2()中，在垃圾回收前，先将变量 a 置为 null，使 byte 数组失去强引用，故垃圾回收可以顺利回收 byte 数组。

对于 localvarGc3()，在进行垃圾回收前，先使局部变量 a 失效，虽然变量 a 已经离开了作用域，但是变量 a 依然存在于局部变量表中，并且也指向这块 byte 数组，故 byte 数组依然无法被回收。

对于 localvarGc4()，在垃圾回收之前，不仅使变量 a 失效，更是申明了变量 c，使变量 c 复用了变量 a 的字，由于变量 a 此时被销毁，故垃圾回收器可以顺利回收 byte 数组。

对于 localvarGc5()，它首先调用了 localvarGc1()，很明显，在 localvarGc1()中并没有释放 byte 数组，但在 localvarGc1()返回后，它的栈帧被销毁，自然也包含了栈帧中的所有局部变量，故 byte 数组失去引用，在 localvarGc5()的垃圾回收中被回收。

读者可以使用参数-XX:+PrintGC 执行上述几个函数，在输出的日志中，可以看到垃圾回收前后堆的大小，进而推断 byte 数组是否被回收。下面的输出是函数 localvarGc4()的运行结果：

```
[Full GC 6746K->376K(15872K), 0.0034589 secs]
```

从日志中可以看到，堆空间从回收前的 6746KB 变为回收后的 376KB，释放了约 6MB 空间。进而可以推断，byte 数组已被回收释放。

2.4.2 操作数栈

操作数栈也是栈帧中重要的内容之一，它主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间。

操作数栈也是一个先进后出的数据结构，只支持入栈和出栈两种操作。许多 Java 字节码指令都需要通过操作数栈进行参数传递。比如 iadd 指令，它就会在操作数栈中弹出两个整数并进行加法计算，计算结果会被入栈，如图 2.10 所示，显示了 iadd 前后操作数栈的变化。

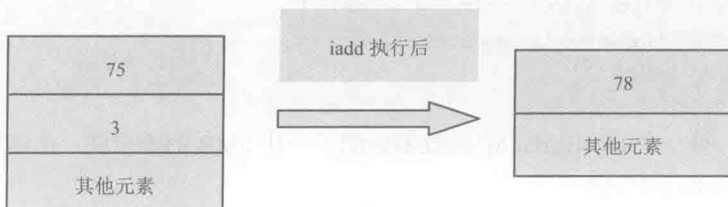


图 2.10 iadd 指令与操作数栈的变化

有关操作数栈和局部变量表使用的详细案例，读者可以参考本书第 11.1 节。

2.4.3 帧数据区

除了局部变量表和操作数栈外，Java 栈帧还需要一些数据来支持常量池解析、正常方法返回和异常处理等。大部分 Java 字节码指令需要进行常量池访问，在帧数据区中保存着访问常量池的指针，方便程序访问常量池。

此外，当函数返回或者出现异常时，虚拟机必须恢复调用者函数的栈帧，并让调用者函数继续执行下去。对于异常处理，虚拟机必须有一个异常处理表，方便在发生异常的时候找到处理异常的代码，因此异常处理表也是帧数据区中重要的一部分。一个典型的异常处理表如下所示：

```
     Exception table:
      from    to  target type
        4    16    19   any
       19    21    19   any
```

它表示在字节码偏移量 4~16 字节可能抛出任意异常，如果遇到异常，则跳转到字节码偏移 19 处执行。当方法抛出异常时，虚拟机就会查找类似的异常表来进行处理，如果无法在异常表中找到合适的处理方法，则会结束当前函数调用，返回调用函数，并在调用函数中抛出相同的异常，并查找调用函数的异常表进行处理。

2.4.4 栈上分配

栈上分配是 Java 虚拟机提供的一项优化技术，它的基本思想是，对于那些线程私有的对象（这里指不可能被其他线程访问的对象），可以将它们打散分配在栈上，而不是分配在堆上。分配在栈上的好处是可以在函数调用结束后自行销毁，而不需要垃圾回收器的介入，从而提高系统的性能。

栈上分配的一个技术基础是进行逃逸分析。逃逸分析的目的是判断对象的作用域是否有可能逃逸出函数体。如下代码显示了一个逃逸的对象：

```
private static User u;
public static void alloc(){
    u=new User();
    u.id=5;
    u.name="geym";
}
```

对象 User u 是类的成员变量，该字段有可能被任何线程访问，因此属于逃逸对象。而以下代码片段显示了一个非逃逸的对象：

```
public static void alloc(){
    User u=new User();
    u.id=5;
    u.name="geym";
}
```

在上述代码中，对象 User 以局部变量的形式存在，并且该对象并没有被 alloc()函数返回，或者出现了任何形式的公开，因此，它并未发生逃逸，所以对于这种情况，虚拟机就有可能将 User 分配在栈上，而不在堆上。

【示例 2-7】下面这个简单的示例显示了对非逃逸对象的栈上分配。

```
public class OnStackTest {
    public static class User{
        public int id=0;
        public String name="";
    }

    public static void alloc(){
        User u=new User();
        u.id=5;
        u.name="geym";
    }
    public static void main(String[] args) throws InterruptedException {
        long b=System.currentTimeMillis();
        for(int i=0;i<100000000;i++){
            alloc();
        }
        long e=System.currentTimeMillis();
        System.out.println(e-b);
    }
}
```

上述代码在主函数中进行了 1 亿次 alloc()调用进行对象创建，由于 User 对象实例需要占据约 16 字节的空间，因此累计分配空间达到将近 1.5GB。如果堆空间小于这个值，就必然会发生 GC。使用如下参数运行上述代码：

```
-server -Xmx10m -Xms10m -XX:+DoEscapeAnalysis -XX:+PrintGC -XX:-UseTLAB
-XX:+EliminateAllocations
```

这里使用参数-server 执行程序，因为在 Server 模式下，才可以启用逃逸分析。参数-XX:+DoEscapeAnalysis 启用逃逸分析，-Xmx10m 指定了堆空间最大为 10MB，显然，如果对象在堆上分配，必然会引起大量的 GC。如果 GC 真的发生了，参数-XX:+PrintGC 将打印 GC 日志。参数-XX:+EliminateAllocations 开启了标量替换（默认打开），允许将对象打散分配在栈上，比如对象拥有 id 和 name 两个字段，那么这两个字段将会被视为两个独立的局部变量进行分配。参数-XX:-UseTLAB 关闭了 TLAB。

程序执行后，完整的输出打印如下：

```
6
```

可以看到，没有任何形式的 GC 输出，程序就执行完毕了。说明在执行过程中，User 对象的分配过程被优化。

如果关闭逃逸分析或者标量替换中任何一个，再次执行程序，就会看到大量的 GC 日志，说明栈上分配依赖逃逸分析和标量替换的实现。

对于大量的零散小对象，栈上分配提供了一种很好的对象分配优化策略，栈上分配速度快，并且可以有效避免垃圾回收带来的负面影响，但由于和堆空间相比，栈空间较小，因此对于大对象无法也不适合在栈上分配。

2.5　类去哪儿了：识别方法区

和 Java 堆一样，方法区是一块所有线程共享的内存区域。它用于保存系统的类信息，比如类的字段、方法、常量池等。方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出错误。

在 JDK 1.6、JDK 1.7 中，方法区可以理解为永久区（Perm）。永久区可以使用参数-XX:PermSize 和-XX:MaxPermSize 指定，默认情况下，-XX:MaxPermSize 为 64MB。一个大的永久区可以保存更多的类信息。如果系统使用了一些动态代理，那么有可能会在运行时生成大量的类，如果这样，就需要设置一个合理的永久区大小，确保不发生永久区内存溢出。

【示例 2-8】下面这段代码使用 CGLIB 库生成大量的动态类。

```
public class PermTest {
    public static void main(String[] args) {
        int i = 0;
        try {
            for (i = 0; i < 100000; i++) {
                CglibBean bean = new CglibBean("geym.zbase.ch2.perm" + i, new
HashMap());
            }
        } catch (Exception e) {
            System.out.println("total create count:" + i);
        }
    }
}
```

限于篇幅有限，这里不再给出 CglibBean 的具体实现。在 CglibBean 的构造函数中，会根据传入参数，动态生成一个类以及一个类的实例对象。任何一个动态产生类的程序都可用于本节的实验。

注意：这里使用 CGLIB 动态产生类，不仅仅是对象实例。由于类的信息（字段、方法、字节码等）保存在方法区，因此，这个操作会占用方法区的空间。

大量的类生成可能导致永久区溢出，使用如下参数运行上述代码：

```
-XX:+PrintGCDetails -XX:PermSize=5M -XX:MaxPermSize=5m
```

这里指定了初始永久区 5MB，最大永久区 5MB，即当 5MB 空间耗尽时，系统将抛出内存溢出。执行程序后，部分输出如下：

```
[Full GC[Tenured: 3014K->2469K(10944K), 0.0165196 secs] 3014K->2469K(15936K), [Perm : 4096K->4094K(4096K)], 0.0165845 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
total create count:1282
Heap
def new generation   total 4992K, used 183K [0x28280000, 0x287e0000, 0x2d7d0000)
eden space 4480K,   4% used [0x28280000, 0x282adcd8, 0x286e0000)
from space 512K,   0% used [0x286e0000, 0x286e0000, 0x28760000)
to   space 512K,   0% used [0x28760000, 0x28760000, 0x287e0000)
tenured generation   total 10944K, used 2469K [0x2d7d0000, 0x2e280000, 0x38280000)
the space 10944K,  22% used [0x2d7d0000, 0x2da39410, 0x2da39600, 0x2e280000)
compacting perm gen  total 4096K, used 4094K [0x38280000, 0x38680000, 0x38680000)
the space 4096K,  99% used [0x38280000, 0x3867fad8, 0x3867fc00, 0x38680000)
ro space 10240K,  44% used [0x38680000, 0x38af73f0, 0x38af7400, 0x39080000)
rw space 12288K,  52% used [0x39080000, 0x396cdd28, 0x396cde00, 0x39c80000)
```

可以看到，在大约 1280 个类产生后，系统发生了内存溢出。扩大-XX:MaxPermSize 的值，那么系统将可以生成更多的动态类，读者可以自行实验。

图 2.11 显示了在 Visual VM 中观察永久区的使用情况。

在 JDK 1.8 中，永久区已经被彻底移除。取而代之的是元数据区，元数据区大小可以使用参数-XX:MaxMetaspaceSize 指定（一个大的元数据区可以使系统支持更多的类），这是一块堆外的直接内存。与永久区不同，如果不指定大小，默认情况下，虚拟机会耗尽所有的可用系统内存。图 2.12 显示了 JDK 1.8 中的元数据区。

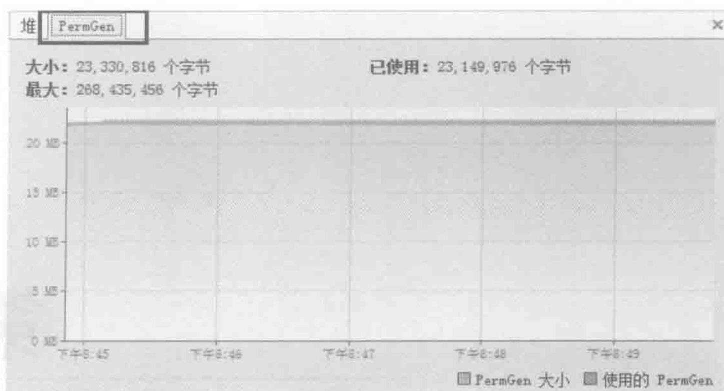


图 2.11　在 Visual VM 中观察永久区

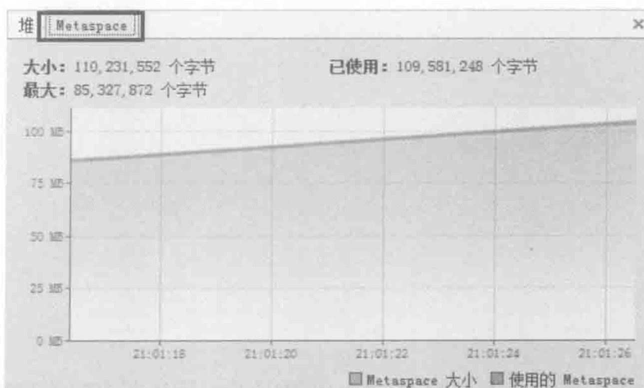


图 2.12　JDK 1.8 的元数据区

如果元数据区发生溢出，虚拟机一样会抛出异常，如下所示：

```
Caused by: java.lang.OutOfMemoryError: Metaspace
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:760)
    ... 9 more
```

2.6　小结

本章主要介绍了 Java 虚拟机的基本结构，其中，对 Java 堆、Java 栈和方法区做了较为完整的说明。同时，也对如何设置 Java 虚拟机参数做了简要说明。对于具体的虚拟机参数，本章并未给出太多信息，但在本书的后续章节，将会逐步介绍一些实用的虚拟机参数。

第 3 章 常用 Java 虚拟机参数

要诊断虚拟机，我们就要学习如何对 Java 虚拟机进行最基本的配置和跟踪。本章将主要介绍一些常用的 Java 虚拟机参数，它们可以对系统进行跟踪和配置，对系统故障诊断、性能优化有着重要的作用。

本章涉及的主要知识点有：

- 跟踪 Java 虚拟机的垃圾回收和类加载等信息。
- 配置 Java 虚拟机的堆空间。
- 配置永久区和 Java 栈。
- 学习虚拟机的服务器和客户端模式。

3.1 一切运行都有迹可循：掌握跟踪调试参数

在虚拟机的运行过程中，如果可以跟踪系统的运行状态，那么对于问题的故障排查会有一

定帮助。为此，虚拟机提供了一些跟踪系统状态的参数，使用给定的参数执行 Java 虚拟机，就可以在系统运行时打印相关日志，用于问题分析。

3.1.1 跟踪垃圾回收——读懂虚拟机日志

Java 的一大特色就是支持自动的垃圾回收（GC），但是有时候，如果垃圾回收频繁出现，或者占用了太长的 CPU 时间，就不得不引起重视。此时，就需要一些跟踪参数来进一步甄别垃圾回收器的效率和效果。

最简单的一个 GC 参数是-XX:+PrintGC，使用这个参数启动 Java 虚拟机后，只要遇到 GC，就会打印日志，如下所示：

```
[GC 4793K->377K(15872K), 0.0006926 secs]
[GC 4857K->377K(15936K), 0.0003595 secs]
[GC 4857K->377K(15936K), 0.0001755 secs]
[GC 4857K->377K(15936K), 0.0001957 secs]
```

该日志显示，一共进行了 4 次 GC，每次 GC 占用一行，在 GC 前，堆空间使用量约为 4MB，GC 后，堆空间使用量为 377KB，当前可用的堆空间总和约为 16MB（15936KB）。最后，显示的是本次 GC 所花费的时间。

如果需要更加详细的信息，则可以使用-XX:+PrintGCDetails 参数。它的输出可能如下：

```
[GC[DefNew: 8704K->1087K(9792K), 0.0665590 secs] 22753K->17720K(31680K),
0.0666180 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]
[GC[DefNew: 9791K->9791K(9792K), 0.0000350 secs][Tenured: 16632K->13533K(21888K),
0.4063120 secs] 26424K->13533K(31680K), [Perm : 2583K->2583K(21248K)], 0.4064710
secs] [Times: user=0.41 sys=0.00, real=0.40 secs]
[GC[DefNew: 8704K->1087K(9792K), 0.0574610 secs] 22237K->16688K(31680K),
0.0575180 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]
Heap
 def new generation   total 9792K, used 4586K [0x00000000f8e00000, 0x00000000f98a0000,
0x00000000f98a0000)
  eden space 8704K,  40% used [0x00000000f8e00000, 0x00000000f916a8e0, 0x00000000f9680000)
  from space 1088K,  99% used [0x00000000f9680000, 0x00000000f978ffe0, 0x00000000f9790000)
  to   space 1088K,   0% used [0x00000000f9790000, 0x00000000f9790000, 0x00000000f98a0000)
 tenured generation   total 21888K, used 15600K [0x00000000f98a0000, 0x00000000fae00000,
0x00000000fae00000)
   the space 21888K,  71% used [0x00000000f98a0000, 0x00000000fa7dc278,
0x00000000fa7dc400, 0x00000000fae00000)
 compacting perm gen  total 21248K, used 2591K [0x00000000fae00000, 0x00000000fc2c0000,
```

```
0x0000000100000000)
   the space 21248K,  12% used [0x00000000fae00000, 0x00000000fb087ca8,
0x00000000fb087e00, 0x00000000fc2c0000)
No shared spaces configured.
```

从这个输出中可以看到，系统经历了 3 次 GC，第 1 次仅为新生代 GC，回收的效果是新生代从回收前的 8MB 左右降低到 1MB。整个堆从 22MB 左右降低到了 17MB。

第 2 次（加粗部分）为 Full GC，它同时回收了新生代、老年代和永久区。日志显示，新生代在这次 GC 中没有释放空间（严格来说，这是 GC 日志的一个小 bug，事实上，在这次 Full GC 完成后，新生代被清空，由于 GC 日志输出时机的关系，各个版本的 JDK 的日志多少有些不太精确的地方，读者需要留意），老年代从 16MB 降低到了 13MB。整个堆大小从 26MB 左右降低为 13MB 左右（这个大小完全与老年代实际大小相等，因此也可以推断，新生代实际上已被清空）。永久区的大小没有变化。在日志的最后，显示了 GC 所花费的时间，其中 user 表示用户态 CPU 耗时，sys 表示系统 CPU 耗时，real 表示 GC 实际经历的时间。

参数-XX:+PrintGCDetails 还会使虚拟机在退出前打印堆的详细信息，详细信息描述了当前堆的各个区间的使用情况。如上输出所示，当前新生代（new generation）总大小为 9792KB，已使用 4586KB。紧跟其后的 3 个 16 进制数字表示新生代的下界、当前上界和上界。

```
[0x00000000f8e00000, 0x00000000f98a0000, 0x00000000f98a0000)
```

使用上界减去下界就能得到当前堆空间的最大值，使用当前上界减去下界，就是当前虚拟机已经为程序分配的空间大小。如果当前上界等于下界，说明当前的堆空间已经没有扩大的可能。在本例中（0x00000000f98a0000-0x00000000f8e00000）/1024=10880KB。这块空间正好等于 eden+from+to 的总和。而可用的新生代 9792KB 为 eden+from(to)的总和，两者的差异原因读者可以参考本书第 4 章。

除了新生代，详细的堆日志中还显示了老年代（tenured generation）和永久区（compacting perm gen）的使用情况，其格式和新生代相同。

如果需要更全面的堆信息，还可以使用参数-XX:+PrintHeapAtGC。它会在每次 GC 前后分别打印堆的信息，就如同-XX:+PrintGCDetails 的最后输出一样。下面就是-XX:+PrintHeapAtGC 的输出样式，限于篇幅，只给出部分输出：

```
{Heap before GC invocations=8 (full 3):
 def new generation   total 8576K, used 8575K [0x32680000, 0x32fc0000, 0x33120000)
  eden space 7680K, 100% used [0x32680000, 0x32e00000, 0x32e00000)
  from space 896K,  99% used [0x32ee0000, 0x32fbffc0, 0x32fc0000)
```

```
 to   space 896K,   0% used [0x32e00000, 0x32e00000, 0x32ee0000)
 tenured generation   total 18880K, used 12353K [0x33120000, 0x34390000, 0x34680000)
 省略部分输出
 [GC[DefNew: 8575K->895K(8576K), 0.0017210 secs] 20929K->14048K(27456K), 0.0017756
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap after GC invocations=9 (full 3):
 def new generation   total 8576K, used 895K [0x32680000, 0x32fc0000, 0x33120000)
  eden space 7680K,   0% used [0x32680000, 0x32680000, 0x32e00000)
  from space 896K,  99% used [0x32e00000, 0x32edffc0, 0x32ee0000)
  to   space 896K,   0% used [0x32ee0000, 0x32ee0000, 0x32fc0000)
 tenured generation   total 18880K, used 13152K [0x33120000, 0x34390000, 0x34680000)
   the space 18880K,  69% used [0x33120000, 0x33df8288, 0x33df8400, 0x34390000)
省略部分输出
}
```

可以看到，在使用-XX:+PrintHeapAtGC 之后，在 GC 日志输出前后，都有详细的堆信息输出，分别表示 GC 回收前和 GC 回收后的堆信息，使用这个参数，可以很好地观察 GC 对堆空间的影响。

如果需要分析 GC 发生的时间，还可以使用-XX:+PrintGCTimeStamps 参数，该参数会在每次 GC 发生时，额外输出 GC 发生的时间，该输出时间为虚拟机启动后的时间偏移量。如下代码表示在系统启动后 0.08 秒、0.088 秒、0.094 秒发生了 3 次 GC。

```
  0.080: [GC0.080: [DefNew: 4416K->512K(4928K), 0.0055792 secs] 4416K->3889K(15872K),
0.0057061 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
  0.088: [GC0.088: [DefNew: 4928K->511K(4928K), 0.0044292 secs] 8305K->7751K(15872K),
0.0045321 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
  0.094: [GC0.094: [DefNew: 4927K->511K(4928K), 0.0044136 secs]0.099: [Tenured:
11238K->11327K(11328K), 0.0113929 secs] 12167K->11750K(16256K), [Perm : 142K->
142K(12288K)], 0.0160228 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
```

由于 GC 会引起应用程序停顿，因此，可能还需要特别关注应用程序的执行时间和停顿时间。使用参数-XX:+PrintGCApplicationConcurrentTime 可以打印应用程序的执行时间，使用参数-XX:+PrintGCApplicationStoppedTime 可以打印应用程序由于 GC 而产生的停顿时间，如下所示：

```
Application time: 0.0026770 seconds
Total time for which application threads were stopped: 0.0091600 seconds
Application time: 0.0039006 seconds
Total time for which application threads were stopped: 0.0024330 seconds
```

如果想跟踪系统内的软引用、弱引用、虚引用和 Finallize 队列，则可以使用打开

-XX:+PrintReferenceGC 开关，结果如下所示：

```
[GC[DefNew[SoftReference, 0 refs, 0.0000212 secs][WeakReference, 7 refs, 0.0000046 secs][FinalReference, 4 refs, 0.0000056 secs][PhantomReference, 0 refs, 0.0000036 secs][JNI Weak Reference, 0.0000056 secs]: 2752K->320K(3072K), 0.0031630 secs] 2752K->2574K(9920K), 0.0031937 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

默认情况下，GC 的日志会在控制台中输出，这不便于后续分析和定位问题。为此，虚拟机允许将 GC 日志以文件的形式输出，可以使用参数-Xloggc 指定。比如使用参数-Xloggc:log/gc.log 启动虚拟机，可以在当前目录下的 log 文件夹下的 gc.log 文件中记录所有的 GC 日志。图 3.1 显示了由-Xloggc 生成的 gc.log 文件。



图 3.1　记录 GC 日志到文件

3.1.2　类加载/卸载的跟踪

Java 程序的运行离不开类的加载，为了更好地理解程序的执行，有时候需要知道系统加载了哪些类。一般情况下，系统加载的类存在于文件系统中，以 jar 的形式打包或者以 class 文件的形式存在，可以直接通过文件系统查看。但是随着动态代理、AOP 等技术的普遍使用，系统也极有可能在运行时动态生成某些类，这些类相对比较隐蔽，无法通过文件系统找到，为此，虚拟机提供的类加载/卸载跟踪参数就显得格外有意义。

可以使用参数-verbose:class 跟踪类的加载和卸载，也可以单独使用参数-XX:+ TraceClassLoading 跟踪类的加载，使用参数-XX:+TraceClassUnloading 跟踪类的卸载。这两类参数是等价的。

【示例 3-1】下面这段代码使用 ASM 动态生成名为 Example 的类，并将其反复加载到系统（有关 ASM 的使用，可以参阅第 9.3 节）。

```
public class UnloadClass  implements Opcodes{
    public static void main(String args[]) throws NoSuchMethodException, SecurityException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {
```

```
03          ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAMES);
04          cw.visit(V1_7, ACC_PUBLIC, "Example", null, "java/lang/Object", null);
05          MethodVisitor mw = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
06          mw.visitVarInsn(ALOAD, 0);
07          mw.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "<init>", "()V");
08          mw.visitInsn(RETURN);
09          mw.visitMaxs(0, 0);
10          mw.visitEnd();
11          mw = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main", "([Ljava/lang/String;)V", null, null);
12          mw.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
13          mw.visitLdcInsn("Hello world!");
14          mw.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");
15          mw.visitInsn(RETURN);
16          mw.visitMaxs(0, 0);
17          mw.visitEnd();
18          byte[] code = cw.toByteArray();
19
20          for(int i=0;i<10;i++){
21              UnloadClassLoader loader = new UnloadClassLoader();
22              Method m=ClassLoader.class.getDeclaredMethod("defineClass", String.class,byte[].class,int.class,int.class);
23              m.setAccessible(true);
24              m.invoke(loader, "Example", code, 0, code.length);
25              m.setAccessible(false);
26              System.gc();
27          }
28      }
29 }
```

上述代码第 3~18 行使用 ASM 生成名为 Example 的类，并将其保存在 code 数组中。在第 20~27 行，使用 ClassLoader 将新生成的类反复加载到系统中，每次循环使用新的 ClassLoader 实例，并在循环结束前进行 Full GC，释放上一次循环加载的类，因此，这一过程会涉及类的加载和卸载。

使用参数-XX:+TraceClassUnloading 和参数-XX:+TraceClassLoading 执行上述代码，跟踪类的加载和卸载过程，部分输出如下：

```
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
省略部分输出
[Loaded sun.reflect.DelegatingMethodAccessorImpl from shared objects file]
[Loaded Example from __JVM_DefineClass__]
[Loaded sun.misc.Cleaner from shared objects file]
[Loaded Example from __JVM_DefineClass__]
[Unloading class Example]
省略部分输出
[Unloading class Example]
```

从这份日志中可以看到，系统首先加载了 java.lang.Object 类，作为所有类的父类，它理应在任何类之前被加载。在日志的后半部分，显示系统对 Example 类先后进行了 10 次加载和 9 次卸载（最后一次加载的类没有机会被卸载）。

注意： 动态类的加载是非常隐蔽的，它们由代码逻辑控制，不出现在文件系统中，跟踪这些类，就需要使用-XX:+TraceClassLoading 等参数来观察系统实际使用的类。

Java 虚拟机还允许研发人员在运行时打印、查看系统中类的分布情况，只要在系统启动时加上-XX:+PrintClassHistogram 参数，然后在 Java 的控制台中按下 Ctrl+Break 组合键，控制台上就会显示当前的类信息柱状图，如下所示：

```
num     #instances         #bytes  class name
----------------------------------------------
   1:        890617      470266000  [B
   2:        890643       21375432  java.util.HashMap$Node
   3:        890608       14249728  java.lang.Long
   4:            13        8389712  [Ljava.util.HashMap$Node;
   5:          2062         371680  [C
   6:           463          41904  java.lang.Class
```

通过这个柱状图信息，可以看到当前系统中占用空间最多的对象，以及其实例数量和空间大小。

3.1.3 系统参数查看

由于目前的 Java 虚拟机支持众多的可配参数，不同的参数可能对系统的执行效果有较大的影响，因此，有必要明确当前系统的实际运行参数。虚拟机提供了一些手段来帮助研发人员获得这些参数。

参数-XX:+PrintVMOptions 可以在程序运行时，打印虚拟机接受到的命令行显式参数。其输出如下所示：

```
VM option '+PrintVMOptions'
VM option '+UseSerialGC'
VM option '+DisableExplicitGC'
```

这说明该虚拟机启动时，命令行明确指定了 UseSerialGC、DisableExplicitGC 两个参数。

参数-XX:+PrintCommandLineFlags 可以打印传递给虚拟机的显式和隐式参数，隐式参数未必是通过命令行直接给出的，它可能是由虚拟机启动时自行设置的，使用-XX:+PrintCommandLineFlags 后，可能的一种输出如下所示：

```
-XX:+DisableExplicitGC -XX:InitialHeapSize=16777216 -XX:MaxHeapSize=268435456
-XX:+PrintCommandLineFlags -XX:-UseLargePagesIndividualAllocation -XX:+UseSerialGC
```

在本例中，-XX: InitialHeapSize、-XX:MaxHeapSize 和-XX:-UseLargePagesIndividualAllocation 并未在命令行显式指定，是由虚拟机自行设置的。

此外，另一个有用的参数是-XX:+PrintFlagsFinal，它会打印所有的系统参数的值，因此，如果需要查看系统的详细参数，则应该使用-XX:+PrintFlagsFinal，开启这个参数后，虚拟机可能会产生多达 500 多行的输出，每一行为一个配置参数和其当前取值，读者如果对虚拟机的各项参数感兴趣，可以打印出这些参数，逐个学习。

3.2　让性能飞起来：学习堆的配置参数

堆空间是 Java 进程的重要组成部分，几乎所有与应用相关的内存空间都和堆有关。本节将主要介绍与堆有关的参数设置，这些参数可以说是 Java 虚拟机中最重要的，也是对程序性能有着重要影响的。

3.2.1　最大堆和初始堆的设置

当 Java 进程启动时，虚拟机就会分配一块初始堆空间，可以使用参数-Xms 指定这块空间的大小。一般来说，虚拟机会尽可能维持在初始堆空间的范围内运行。但是如果初始堆空间耗尽，虚拟机将会对堆空间进行扩展，其扩展上限为最大堆空间，最大堆空间可以使用参数-Xmx 指定。

【示例 3-2】下面通过一个简单的示例，说明最大堆、初始堆以及系统可用内存的含义和彼

此之间的关系。

```
01 public class HeapAlloc {
02     public static void main(String[] args) {
03         System.out.print("maxMemory=");
04         System.out.println(Runtime.getRuntime().maxMemory()+" bytes");
05         System.out.print("free mem=");
06         System.out.println(Runtime.getRuntime().freeMemory()+" bytes");
07         System.out.print("total mem=");
08         System.out.println(Runtime.getRuntime().totalMemory()+" bytes");
09
10         byte[] b=new byte[1*1024*1024];
11         System.out.println("分配了 1M 空间给数组");
12
13         System.out.print("maxMemory=");
14         System.out.println(Runtime.getRuntime().maxMemory()+" bytes");
15         System.out.print("free mem=");
16         System.out.println(Runtime.getRuntime().freeMemory()+" bytes");
17         System.out.print("total mem=");
18         System.out.println(Runtime.getRuntime().totalMemory()+" bytes");
19
20         b=new byte[4*1024*1024];
21         System.out.println("分配了 4M 空间给数组");
22
23         System.out.print("maxMemory=");
24         System.out.println(Runtime.getRuntime().maxMemory()+" bytes");
25         System.out.print("free mem=");
26         System.out.println(Runtime.getRuntime().freeMemory()+" bytes");
27         System.out.print("total mem=");
28         System.out.println(Runtime.getRuntime().totalMemory()+" bytes");
29     }
30 }
```

上述代码首先在第 3~8 行打印了基本的系统信息，包括最大可用内存、当前空闲内存和当前总内存。接着，在第 10 行申请了 1MB 内存空间，显然，这块空间将在堆上分配。在第 13~18 行，同样打印了最大可用内存、当前空闲内存和当前总内存。接着，在第 20 行再次申请了 4MB 空间。最后，同样打印了这 3 个参数。

根据前文的介绍很容易让人想到，这里的最大可用内存就是指-Xmx 的取值，当前总内存应该不小于-Xms 的设定，因为当前总内存总是在-Xms 和-Xmx 之间，从-Xms 开始根据需要向上增长。而当前空闲内存应该是当前总内存减去当前已经使用的空间。

这样的理解差不多是正确的，之所以使用“差不多”一词，是因为这样理解并不是十分全面的。使用如下参数执行程序，很快就能发现中间的偏差。

```
-Xmx20m -Xms5m -XX:+PrintCommandLineFlags -XX:+PrintGCDetails -XX:+UseSerialGC
```

程序的输出如下所示：

```
-XX:InitialHeapSize=5242880 -XX:MaxHeapSize=20971520 -XX:+PrintCommandLineFlags
-XX:+PrintGCDetails -XX:-UseLargePagesIndividualAllocation -XX:+UseSerialGC
maxMemory=20316160 bytes
free mem=4553832 bytes
total mem=5111808 bytes
[GC[DefNew: 544K->128K(1536K), 0.0011590 secs] 544K->377K(4992K), 0.0012395
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
分配了 1M 空间给数组
maxMemory=20316160 bytes
free mem=3619072 bytes
total mem=5111808 bytes
[GC[DefNew: 1208K->0K(1536K), 0.0010289 secs][Tenured: 1401K->1401K(3456K),
0.0030155 secs] 1457K->1401K(4992K), [Perm : 143K->143K(12288K)], 0.0041523 secs]
[Times: user=0.02 sys=0.00, real=0.01 secs]
分配了 4M 空间给数组
maxMemory=20316160 bytes
free mem=3795728 bytes
total mem=9441280 bytes
Heap
 def new generation   total 1664K, used 46K [0x33280000, 0x33440000, 0x33920000)
  eden space 1536K,   3% used [0x33280000, 0x3328ba68, 0x33400000)
  from space 128K,   0% used [0x33400000, 0x33400000, 0x33420000)
  to   space 128K,   0% used [0x33420000, 0x33420000, 0x33440000)
 tenured generation   total 7556K, used 5497K [0x33920000, 0x34081000, 0x34680000)
   the space 7556K,  72% used [0x33920000, 0x33e7e5d0, 0x33e7e600, 0x34081000)
 compacting perm gen  total 12288K, used 143K [0x34680000, 0x35280000, 0x38680000)
   the space 12288K,   1% used [0x34680000, 0x346a3db0, 0x346a3e00, 0x35280000)
    ro space 10240K,  44% used [0x38680000, 0x38af73f0, 0x38af7400, 0x39080000)
    rw space 12288K,  52% used [0x39080000, 0x396cdd28, 0x396cde00, 0x39c80000)
```

可以看到，当前的最大内存由-XX:MaxHeapSize=20971520 指定，它正好是 20*1024*1024=20971520 字节。而打印的最大可用内存仅仅为 20316160 字节，比设定值略少。这是因为分配给堆的内存空间和实际可用的内存空间并非一个概念。由于垃圾回收的需要，虚拟机会对堆空

间进行分区管理，不同的区域采用不同的回收算法，一些算法会使用空间换时间的策略工作，因此会存在可用内存的损失，详细算法可以参见第 4.2.3 节，这里不予展开讨论。最终的结果就是实际可用内存会浪费大小等于 from/to 的空间。因此，实际最大可用内存为-Xmx 的值减去 from 的大小。根据堆的详细信息，可以看到，from 的大小为 0x33420000-0x33400000=0x20000=131072 字节。但很不幸，读者应该会发现 20971520-131072=20840448 字节，与实际值 20316160 依然存在偏差。这个偏差是由于虚拟机内部并没有直接使用新生代 from/to 的大小，而是进一步对它们做了对齐操作。对于串行 GC 的情况，虚拟机使用以下方法估算 from/to 的大小，并进行了对齐。

```
#define align_size_down_(size, alignment) ((size) & ~((alignment) - 1))

inline intptr_t align_size_down(intptr_t size, intptr_t alignment) {
  return align_size_down_(size, alignment);
}

size_t compute_survivor_size(size_t gen_size, size_t alignment) const {
    size_t n = gen_size / (SurvivorRatio + 2);
    return n > alignment ? align_size_down(n, alignment) : alignment;
  }
```

上述代码中的 alignment 变量，在非 ARM 平台上为 1<<16，即 2^{16}，参数 gen_size 表示新生代的总大小，SurvivorRatio 默认值为 8，表示幸存代比例。在本例中，使用 compute_survivor_size 估算并对齐后的 from/to 区间大小为：

$$n= gen_size / (SurvivorRatio + 2)= 6946816/10=694681$$
$$(size) \& \sim((alignment) - 1)= 694681 \& \sim(2^{16}-1)$$
$$= 0xA9999\& 0xFFFFFFFFFFFE0000=0x A0000= 655360$$

使用 20971520 减去 655360 后，得出实际可用最大堆空间为 20316160 字节，与输出值吻合。由此，可以解释实际可用堆大小与-Xmx 配置值之间的偏差。

程序运行初期，打印显示，空闲内存约为 4.5MB，总内存约为 5MB。在进行了 1MB 内存分配后，最大可用内存不变（理论上它永远不会变），总内存依然为 5MB，而空闲内存约为 3.5MB，这与分配了 1MB 内存有关。之后，程序申请了 4MB 内存，由于剩余可用内存已经不能满足需要，因此堆空间进行扩展，扩展后总内存约为 10MB，空闲内存约为 3.5MB。

提示：在实际工作中，也可以直接将初始堆-Xms 与最大堆-Xmx 设置相等。这样的好处是可以减少程序运行时进行的垃圾回收次数，从而提高程序的性能。

3.2.2 新生代的配置

参数-Xmn 可以用于设置新生代的大小。设置一个较大的新生代会减小老年代的大小，这个参数对系统性能以及 GC 行为有很大的影响。新生代的大小一般设置为整个堆空间的 1/3 到 1/4 左右。

参数-XX:SurvivorRatio 用来设置新生代中 eden 空间和 from/to 空间的比例关系，它的含义如下：

```
-XX:SurvivorRatio=eden/from=eden/to
```

【示例 3-3】考察以下这段简单的 Java 程序，它连续向系统请求 10MB 空间（每次申请 1MB）。

```
public class NewSizeDemo {
    public static void main(String[] args) {
        byte[] b=null;
        for(int i=0;i<10;i++)
            b=new byte[1*1024*1024];
    }
}
```

使用不同的堆分配参数执行这段程序，虚拟机的行为表现受到堆空间分配的影响，运行过程不尽相同。下面分别使用不同的参数执行，读者可以跟随本书一块来实战演习下。

① 使用-Xmx20m -Xms20m -Xmn1m　-XX:SurvivorRatio=2 -XX:+PrintGCDetails 运行上述 Java 程序，便会有以下输出：

```
[GC[DefNew: 512K->256K(768K), 0.0013368 secs] 512K->370K(20224K), 0.0013772
secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 768K, used 303K [0x33060000, 0x33160000, 0x33160000)
  eden space 512K,   9% used [0x33060000, 0x3306bfb0, 0x330e0000)
  from space 256K, 100% used [0x33120000, 0x33160000, 0x33160000)
  to   space 256K,   0% used [0x330e0000, 0x330e0000, 0x33120000)
 tenured generation   total 19456K, used 10354K [0x33160000, 0x34460000, 0x34460000)
   the space 19456K,  53% used [0x33160000, 0x33b7c9f0, 0x33b7ca00, 0x34460000)
 compacting perm gen  total 12288K, used 142K [0x34460000, 0x35060000, 0x38460000)
   the space 12288K,   1% used [0x34460000, 0x34483a90, 0x34483c00, 0x35060000)
    ro space 10240K,  44% used [0x38460000, 0x388d73f0, 0x388d7400, 0x38e60000)
    rw space 12288K,  52% used [0x38e60000, 0x394add28, 0x394ade00, 0x39a60000)
```

这里 eden 与 from 的比值为 2 比 1，故 eden 区为 512KB。总可用的新生代为 512KB+256KB=768KB，而新生代总大小为 512KB+256KB+256KB =1024KB=1MB。

由于 eden 区无法容纳任何一个程序中分配的 1MB 数组，故触发了一次新生代 GC，对 eden 区进行了部分回收，同时，这个偏小的新生代无法为 1MB 数组预留空间，故所有的数组都分配在老年代，老年代最终占用 10354KB 空间。

② 使用参数-Xmx20m -Xms20m -Xmn7m -XX:SurvivorRatio=2 -XX:+PrintGCDetails 运行上述程序，将新生代扩大为 7MB，则输出如下：

```
[GC[DefNew: 2583K->1401K(5376K), 0.0015520 secs] 2583K->1401K(18688K), 0.0015898 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC[DefNew: 4588K->1024K(5376K), 0.0011594 secs] 4588K->1401K(18688K), 0.0011865 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC[DefNew: 4124K->1024K(5376K), 0.0005396 secs] 4501K->1401K(18688K), 0.0005681 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 5376K, used 3162K [0x33060000, 0x33760000, 0x33760000)
  eden space 3584K,  59% used [0x33060000, 0x33276960, 0x333e0000)
  from space 1792K,  57% used [0x335a0000, 0x336a0010, 0x33760000)
  to   space 1792K,   0% used [0x333e0000, 0x333e0000, 0x335a0000)
 tenured generation   total 13312K, used 377K [0x33760000, 0x34460000, 0x34460000)
   the space 13312K,   2% used [0x33760000, 0x337be5a8, 0x337be600, 0x34460000)
 compacting perm gen  total 12288K, used 142K [0x34460000, 0x35060000, 0x38460000)
   the space 12288K,   1% used [0x34460000, 0x34483a90, 0x34483c00, 0x35060000)
    ro space 10240K,  44% used [0x38460000, 0x388d73f0, 0x388d7400, 0x38e60000)
    rw space 12288K,  52% used [0x38e60000, 0x394add28, 0x394ade00, 0x39a60000)
```

在这个参数下，由于 eden 区有足够的空间，因此所有的数组都首先分配在 eden 区。但 eden 区并不足以预留全部 10MB 的空间，故在程序运行期间，出现了 3 次新生代 GC。由于程序中每申请一次空间，也同时废弃了上一次申请的内存（上次申请的内存失去了引用），故在新生代 GC 中，有效回收了这些失效的内存。最终结果是：所有的内存分配都在新生代进行，通过 GC 保证了新生代有足够的空间，而老年代没有为这些数组预留任何空间，只是在 GC 过程中，部分新生代对象晋升到老年代。

③ 使用参数-Xmx20m -Xms20m -Xmn15m -XX:SurvivorRatio=8 -XX:+PrintGCDetails 运行上述程序，得到的输出如下：

```
Heap
 def new generation   total 13824K, used 11223K [0x33060000, 0x33f60000, 0x33f60000)
  eden space 12288K,  91% used [0x33060000, 0x33b55f40, 0x33c60000)
  from space 1536K,   0% used [0x33c60000, 0x33c60000, 0x33de0000)
  to   space 1536K,   0% used [0x33de0000, 0x33de0000, 0x33f60000)
```

```
  tenured generation   total 5120K, used 0K [0x33f60000, 0x34460000, 0x34460000)
   the space 5120K,   0% used [0x33f60000, 0x33f60000, 0x33f60200, 0x34460000)
  compacting perm gen  total 12288K, used 142K [0x34460000, 0x35060000, 0x38460000)
   the space 12288K,   1% used [0x34460000, 0x34483a90, 0x34483c00, 0x35060000)
    ro space 10240K,  44% used [0x38460000, 0x388d73f0, 0x388d7400, 0x38e60000)
    rw space 12288K,  52% used [0x38e60000, 0x394add28, 0x394ade00, 0x39a60000)
```

在这次执行中，由于新生代使用 15MB 空间，其中 eden 区占用了 12288KB，完全满足 10MB 数组的分配，因此所有的分配行为都在 eden 直接进行，且没有触发任何 GC 行为。因此 from/to 和老年代 tenured 的使用率都为 0。

由此可见，不同的堆分布情况，对系统执行会产生一定影响。在实际工作中，应该根据系统的特点做合理的设置，基本策略是：尽可能将对象预留在新生代，减少老年代 GC 的次数（在本例中的第一种情况，对象都分配在老年代，显然为后续的老年代 GC 埋下了伏笔）。

除了可以使用参数-Xmn 指定新生代的绝对大小外，还可以使用参数-XX:NewRatio 来设置新生代和老年代的比例。它的含义如下：

```
-XX:NewRatio=老年代/新生代
```

④ 使用参数-Xmx20M -Xms20M -XX:NewRatio=2　-XX:+PrintGCDetails 运行上述代码，输出如下：

```
  [GC[DefNew:  4647K->377K(6144K),  0.0016252  secs]  4647K->1401K(19840K),
0.0016728 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
  [GC[DefNew:  5669K->0K(6144K),  0.0011792  secs]  6693K->2425K(19840K),
0.0012087 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
  Heap
  def new generation   total 6144K, used 1134K [0x33060000, 0x33700000, 0x33700000)
   eden space 5504K,  20% used [0x33060000, 0x3317b8c8, 0x335c0000)
   from space 640K,   0% used [0x335c0000, 0x335c0088, 0x33660000)
   to   space 640K,   0% used [0x33660000, 0x33660000, 0x33700000)
  tenured generation   total 13696K, used 2425K [0x33700000, 0x34460000, 0x34460000)
   the space 13696K,  17% used [0x33700000, 0x3395e540, 0x3395e600, 0x34460000)
  compacting perm gen  total 12288K, used 142K [0x34460000, 0x35060000, 0x38460000)
   the space 12288K,   1% used [0x34460000, 0x34483a90, 0x34483c00, 0x35060000)
    ro space 10240K,  44% used [0x38460000, 0x388d73f0, 0x388d7400, 0x38e60000)
    rw space 12288K,  52% used [0x38e60000, 0x394add28, 0x394ade00, 0x39a60000)
```

此时，因为堆大小为 20MB。新生代和老年代的比为 1 比 2。故新生代大小为 20MB*1/3=6MB 左右，老年代为 13MB 左右。由于在新生代 GC 时，from/to 空间不足以容纳任何一个 1MB 数

组，影响了新生代的正常回收，故在新生代回收时需要老年代进行空间担保。因此，导致两个 1MB 数组进入老年代（在新生代 GC 时，尚有 1MB 数组幸存，理应进入 from/to，而 from/to 只有 640KB，不足以容纳）。

注意：-XX:SurvivorRatio 可以设置 eden 区与 survivor 区的比例。-XX:NewRatio 可以设置老年代与新生代的比例。

图 3.2 显示了几个重要的堆分配参数的含义。

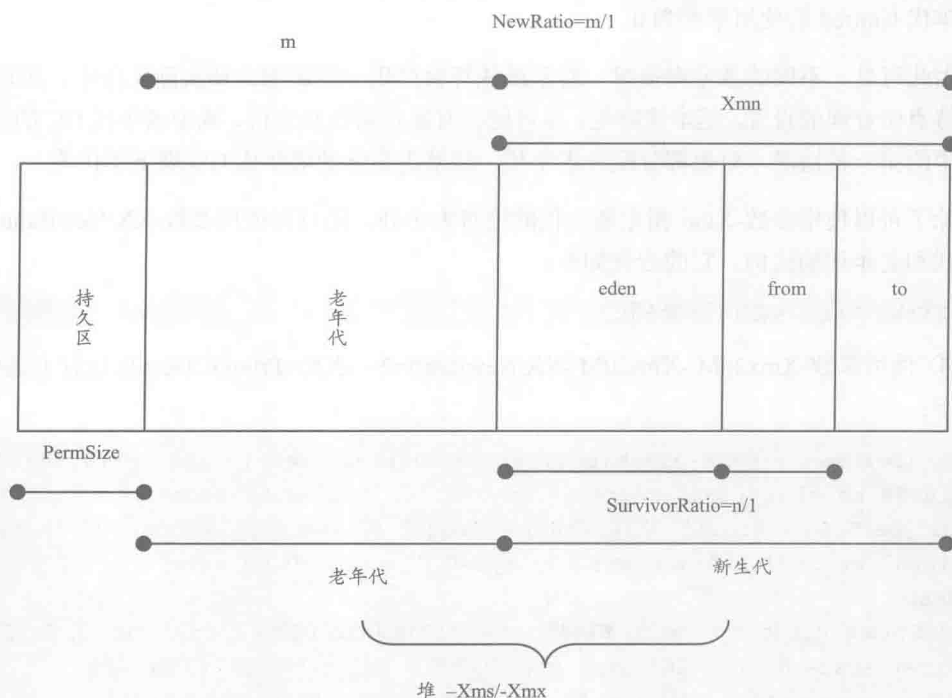


图 3.2 堆的分配参数示意图

3.2.3 堆溢出处理

在 Java 程序的运行过程中，如果堆空间不足，则有可能抛出内存溢出错误（Out Of Memory），简称为 OOM。如下文字显示了典型的堆内存溢出：

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
 at geym.zbase.ch3.heap.DumpOOM.main(DumpOOM.java:20)
```

一旦发生这类问题，系统就会被迫退出。如果发生在生产环境，可能会引起严重的业务中断。为了能够不断改善系统，避免或减少这类错误的发生，需要在发生错误时，获得尽可能多的现场信息，以帮助研发人员排查现场问题。Java 虚拟机提供了参数-XX:+HeapDumpOnOutOfMemoryError，使用该参数，可以在内存溢出时导出整个堆信息。和它配合使用的还有-XX:HeapDumpPath，可以指定导出堆的存放路径。

【示例 3-4】以下代码合计分配了 25MB 内存空间。

```
public class DumpOOM {
    public static void main(String[] args) {
        Vector v=new Vector();
        for(int i=0;i<25;i++)
            v.add(new byte[1*1024*1024]);
    }
}
```

使用如下参数执行上述代码：

```
-Xmx20m -Xms5m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:/a.dump
```

显然 20MB 堆空间不足以容纳 25MB 内存，系统发生内存溢出，在发生错误后，控制台输出如下：

```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to d:/a.dump ...
Heap dump file created [23067302 bytes in 0.160 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
 at geym.zbase.ch3.heap.DumpOOM.main(DumpOOM.java:19)
```

可以看到，虚拟机将当前的堆导出，并保存到 D:/a.dump 文件下。使用 MAT 等工具打开该文件进行分析，如图 3.3 所示，可以很容易地找到这些 byte 数组和保存它们的 Vector 对象实例。有关 MAT 等工具的使用，可以参阅第 7 章。

除了在发生 OOM 时可以导出堆信息外，虚拟机还允许在发生错误时执行一个脚本文件。该文件可以用于奔溃程序的自救、报警或者通知，也可以帮助开发人员获得更多的系统信息，如完整的线程转存（即 Thread Dump 或者 Core Dump）文件。

这里给出一个在发生 OOM 时导出线程转存的例子。准备 printstack.bat 脚本如下：

```
D:/tools/jdk1.7_40/bin/jstack -F %1 > D:/a.txt
```

以上脚本将会导出给定 Java 虚拟机进程的线程信息，并保存在 D:/a.txt 文件中。

java.util.Vector @ 0x3395c490	24	18,874,776
<class> class java.util.Vector @ 0x396:	16	16
elementData java.lang.Object[20] @ 0	96	18,874,752
<class> class java.lang.Object[] @ (	0	0
[13] byte[1048576] @ 0x33280000	1,048,592	1,048,592
[14] byte[1048576] @ 0x33380010	1,048,592	1,048,592
[15] byte[1048576] @ 0x33480020	1,048,592	1,048,592
[16] byte[1048576] @ 0x33580030	1,048,592	1,048,592
[17] byte[1048576] @ 0x33680040	1,048,592	1,048,592
[0] byte[1048576] @ 0x3397acf0 ...	1,048,592	1,048,592
[1] byte[1048576] @ 0x33a7b3b0 .	1,048,592	1,048,592
[2] byte[1048576] @ 0x33b7b3c0 ..	1,048,592	1,048,592
[3] byte[1048576] @ 0x33c7b3d0 ..	1,048,592	1,048,592
[4] byte[1048576] @ 0x33d7b3e0 .	1,048,592	1,048,592
[5] byte[1048576] @ 0x33e7b3f0 ...	1,048,592	1,048,592
[6] byte[1048576] @ 0x33f7b400 ...	1,048,592	1,048,592
[7] byte[1048576] @ 0x3407b498 ..	1,048,592	1,048,592
[8] byte[1048576] @ 0x3417b4a8 ...	1,048,592	1,048,592
[9] byte[1048576] @ 0x3427b4b8 .	1,048,592	1,048,592
[10] byte[1048576] @ 0x3437b4c8	1,048,592	1,048,592
[11] byte[1048576] @ 0x3447b4d8	1,048,592	1,048,592
[12] byte[1048576] @ 0x3457b548	1,048,592	1,048,592
Σ **Total: 19 entries**		
Σ **Total: 2 entries**		

图 3.3　MAT 查看堆文件

使用如下参数执行上述代码：

```
-Xmx20m -Xms5m "-XX:OnOutOfMemoryError=D:/tools/jdk1.7_40/bin/printstack.bat %p"
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:/a.dump
```

在程序异常退出时，系统 D 盘下会生成新文件 a.txt，里面保存着线程转存信息。本例中，文件路径“D:/tools/jdk1.7_40”为笔者的 JDK 安装目录，读者可以替换成自己的 JAVA_HOME 目录，进行尝试。

3.3 别让性能有缺口：了解非堆内存的参数配置

除了堆内存外，虚拟机还有一些内存用于方法区、线程栈和直接内存的使用。它们与堆内存是相对独立的。虽然和堆内存相比，这些内存空间和应用程序本身可能关系不那么密切，但是从系统层面上看，有效、合理地配置这些内存参数，对系统性能和稳定性也有着重要的作用。

3.3.1 方法区配置

方法区主要存放类的元信息。

在 JDK 1.6 和 JDK 1.7 等版本中，可以使用-XX:PermSize 和-XX:MaxPermSize 配置永久区大小。其中-XX:PermSize 表示初始的永久区大小，-XX:MaxPermSize 表示最大永久区。

在 JDK 1.8 中，永久区被彻底移除，使用了新的元数据区存放类的元数据。默认情况下，元数据区只受系统可用内存的限制，但依然可以使用参数-XX:MaxMetaspaceSize 指定永久区的最大可用值。有关方法区的详细使用和配置，可以参考第 2.5 节。

3.3.2 栈配置

栈是每个线程私有的内存空间。在 Java 虚拟机中可以使用-Xss 参数指定线程的栈大小。由于在第 2.4 节中已经详细介绍了栈的配置和使用，这里不再展开。

3.3.3 直接内存配置

直接内存也是 Java 程序中非常重要的组成部分，特别是在 NIO 被广泛使用后，直接内存的使用也变得非常普遍。直接内存跳过了 Java 堆，使 Java 程序可以直接访问原生堆空间，因此，从一定程度上加快了内存空间的访问速度。但是，武断地认为使用直接内存一定可以提高内存访问速度也是不正确的。

最大可用直接内存可以使用参数-XX:MaxDirectMemorySize 设置，如不设置，默认值为最大堆空间，即-Xmx。当直接内存使用量达到-XX:MaxDirectMemorySize 时，就会触发垃圾回收，如果垃圾回收不能有效释放足够空间，直接内存溢出依然会引起系统的 OOM。

【示例 3-5】一般来说，直接内存的访问速度（读或者写）会快于堆内存。下面的代码统计了对直接内存和堆内存的读写速度。

```
public class AccessDirectBuffer {
    public void directAccess(){
        long starttime=System.currentTimeMillis();
        ByteBuffer b=ByteBuffer.allocateDirect(500);
        for(int i=0;i<100000;i++){
            for(int j=0;j<99;j++)
                b.putInt(j);
            b.flip();
            for(int j=0;j<99;j++)
```

```
                b.getInt();
            b.clear();
        }
        long endtime=System.currentTimeMillis();
        System.out.println("testDirectWrite:"+(endtime-starttime));
    }

    public void bufferAccess() {
        long starttime=System.currentTimeMillis();
        ByteBuffer b=ByteBuffer.allocate(500);
        for(int i=0;i<100000;i++){
            for(int j=0;j<99;j++)
                b.putInt(j);
            b.flip();
            for(int j=0;j<99;j++)
                b.getInt();
            b.clear();
        }
        long endtime=System.currentTimeMillis();
        System.out.println("testBufferWrite:"+(endtime-starttime));
    }

    public static void main(String[] args) {
        AccessDirectBuffer alloc=new AccessDirectBuffer();
        alloc.bufferAccess();
        alloc.directAccess();

        alloc.bufferAccess();
        alloc.directAccess();
    }
}
```

上述代码中，bufferAccess()方法进行堆内存读写，并进行耗时统计，directAccess()对直接内存进行读写，并进行耗时统计。在笔者的计算机上，上述代码输出如下：

```
testBufferWrite:414
testDirectWrite:232
testBufferWrite:398
testDirectWrite:238
```

程序中，对 bufferAccess()和 directAccess()方法分别进行了两次调用，第一次视为热身代码，这里忽略其输出，只关注第 2 次调用的输出结果。从结果可以看出，直接内存的访问比堆内存

快约 40%。若使用-server 参数执行上述代码（笔者的环境中默认为-client 模式），输出如下：

```
testBufferWrite:95
testDirectWrite:35
testBufferWrite:101
testDirectWrite:13
```

可以看到，直接内存被 Server 优化后，比堆内存提升了将近一个数量级。

虽然在访问读写上直接内存有较大的优势，但是在内存空间申请时，直接内存毫无优势可言。参考以下代码：

```
public class AllocDirectBuffer {
    public void directAllocate(){
        long starttime=System.currentTimeMillis();
        for(int i=0;i<200000;i++){
           ByteBuffer b=ByteBuffer.allocateDirect(1000);
        }
        long endtime=System.currentTimeMillis();
        System.out.println("directAllocate:"+(endtime-starttime));
    }

    public void bufferAllocate() {
        long starttime=System.currentTimeMillis();
        for(int i=0;i<200000;i++){
           ByteBuffer b=ByteBuffer.allocate(1000);
        }
        long endtime=System.currentTimeMillis();
        System.out.println("bufferAllocate:"+(endtime-starttime));
    }

    public static void main(String[] args) {
        AllocDirectBuffer alloc=new AllocDirectBuffer();
        alloc.bufferAllocate();
        alloc.directAllocate();

        alloc.bufferAllocate();
        alloc.directAllocate();
    }
}
```

在笔者的环境中，其执行结果如下：

```
bufferAllocate:68
```

```
directAllocate:306
bufferAllocate:89
directAllocate:279
```

可以看到，在申请内存空间时，堆空间的速度远远高于直接内存。

由此，可以得出结论：直接内存适合申请次数较少、访问较频繁的场合。如果内存空间本身需要频繁申请，则并不适合使用直接内存。

3.4 Client 和 Server 二选一：虚拟机的工作模式

目前的 Java 虚拟机支持 Client 和 Server 两种运行模式。使用参数-client 可以指定使用 Client 模式，使用参数-server 可以指定使用 Server 模式。默认情况下，虚拟机会根据当前计算机系统环境自动选择运行模式。使用-version 参数可以查看当前的模式，如下所示：

```
./java -version
java version "1.7.0_40"
Java(TM) SE Runtime Environment (build 1.7.0_40-b43)
Java HotSpot(TM) Client VM (build 24.0-b56, mixed mode, sharing)
```

使用-server 参数后，就可以得到如下输出：

```
./java -server -version
java version "1.7.0_40"
Java(TM) SE Runtime Environment (build 1.7.0_40-b43)
Java HotSpot(TM) Server VM (build 24.0-b56, mixed mode)
```

与 Client 模式相比，Server 模式的启动比较慢，因为 Server 模式会尝试收集更多的系统性能信息，使用更复杂的优化算法对程序进行优化。因此，当系统完全启动并进入运行稳定期后，Server 模式的执行速度会远远快于 Client 模式。所以，对于后台长期运行的系统，使用-server 参数启动对系统的整体性能可以有不小的帮助。但对于用户界面程序，运行时间不长，又追求启动速度，Client 模式也是不错的选择。

从发展趋势上看，未来 64 位系统必然会逐步取代 32 位系统，而在 64 位系统中虚拟机更倾向于使用 Server 模式运行。

虚拟机在 Server 模式和 Client 模式下的各种参数可能会有很大不同，读者如果需要查看给定参数的默认值，可以使用-XX:+PrintFlagsFinal 参数。这里以 JIT 编译阈值和最大堆为例，展示 Client 模式和 Server 模式下两者的区别。

对于 Client 模式，参数如下：

```
./java -XX:+PrintFlagsFinal -client -version  |grep -E ' CompileThreshold|
MaxHeapSize'
    intx CompileThreshold                          = 1500          {pd product}
    uintx MaxHeapSize                             := 268435456     {product}
java version "1.7.0_40"
Java(TM) SE Runtime Environment (build 1.7.0_40-b43)
Java HotSpot(TM) Client VM (build 24.0-b56, mixed mode, sharing)
```

对于 Server 模式，参数如下：

```
./java -XX:+PrintFlagsFinal -server -version  |grep -E ' CompileThreshold|
MaxHeapSize'
     intx CompileThreshold                          = 10000        {pd product}
    uintx MaxHeapSize                              := 1073741824   {product}
java version "1.7.0_40"
Java(TM) SE Runtime Environment (build 1.7.0_40-b43)
Java HotSpot(TM) Server VM (build 24.0-b56, mixed mode)
```

可以看到，在 Client 模式下，CompileThreshold 的默认值为 1500，即函数被调用 1500 次后，会进行 JIT 编译（有关 JIT 编译的更多细节请参阅第 11 章）。而在 Server 模式下，这个数值为 10000。因此，Server 模式下系统更有可能解释执行。而一旦进行编译，Server 模式的优化效果会好于 Client 模式。其次，对于系统最大堆，在 Client 模式下为约 256MB，而在 Server 模式下约为 1GB。对于其他参数，读者可以使用类似的方式进行查找比较。

3.5　小结

本章主要介绍了一些常用的 Java 虚拟机参数，如垃圾回收跟踪参数、类加载跟踪参数等。同时也详细介绍了堆空间的配置方法以及方法区、Java 栈和直接内存的配置。此外，本章还介绍了系统发生内存溢出错误后的信息获取和补救方法。最后，简要介绍了虚拟机的 Server 和 Client 运行模式。

第4章 垃圾回收概念与算法

垃圾回收是Java体系最重要的组成部分之一。和C/C++的手工内存管理不同，Java虚拟机提供了一套全自动的内存管理方案，尽可能地减少开发人员在内存资源管理方面的工作量。要掌握这套内存管理方案，我们就必须理解垃圾回收器以怎样的原理工作，本章就来介绍垃圾回收器的基本算法和工作方式。

本章涉及的主要知识点有：

- 了解什么是垃圾回收。
- 学习几种常用的垃圾回收算法。
- 掌握可触及性的概念。
- 理解Stop-The-World（STW）。

4.1 内存管理清洁工：认识垃圾回收

谈到垃圾回收（Garbage Collection，简称GC），需要先澄清什么是垃圾。类比日常生活中的垃圾，我们会把它们丢入垃圾桶，然后倒掉。GC 中的垃圾，特指存在于内存中的、不会再

被使用的对象，而“回收”，也相当于把垃圾桶“倒掉”。这样房间里或者内存空间里就会有空闲的区域被腾出来。如果不及时对内存中的垃圾进行清理，那么，这些垃圾对象所占的内存空间会一直保留到应用程序结束，被保留的空间无法被其他对象使用。如果大量不会被使用的对象一直占着空间不放，需要内存空间时，就无法使用这些被垃圾对象占用的内存，从而有可能导致内存溢出。因此，对内存空间的管理来说，识别和清理垃圾对象是至关重要的。

在早期的 C/C++时代，垃圾回收基本上是手工进行的。开发人员可以使用 new 关键字进行内存申请，并使用 delete 关键字进行内存释放。比如以下代码：

```
MibBridge *pBridge = new cmBaseGroupBridge();
if (pBridge->Register(kDestroy) != NO_ERROR)
  delete pBridge;
```

上述代码首先使用 new 关键字创建了一个名为 cmBaseGroupBridge 的对象，并进行注册（Register），如果注册失败，则表示对象不会再被使用，因此使用 delete 释放该对象所占的内存区域。可以看到，内存的释放是由开发人员显式指定的，这种方式可以灵活控制内存释放的时间，但是会给开发人员带来很大的管理负担（一个系统中内存申请和释放可能是极其频繁的）。倘若有一处内存区间由于程序员编码的问题忘记被回收，那么就会产生内存泄漏，垃圾对象永远无法被清除，随着系统运行时间的不断增长，垃圾对象所耗内存可能持续上升，直到内存溢出。

为了将程序员从繁重的内存管理中释放出来，可以更专心地专注于业务开发，就需要一个垃圾回收机制来帮助开发人员在合适的时间，进行垃圾对象的释放。因此，在有了垃圾回收机制后，上述代码块极有可能变成这样：

```
MibBridge *pBridge = new cmBaseGroupBridge();
pBridge->Register(kDestroy);
```

此时可以看到，开发人员只需要关注内存的申请，而内存的释放可以由系统自动识别和完成。

垃圾回收并不是 Java 虚拟机独创的，早在 20 世纪 60 年代，垃圾回收就已经被 Lisp 语言所使用。现在，除了 Java 以外，C#、Python 等语言都使用了垃圾回收的思想。可以说，这种自动化的内存管理方式已经成为现代开发语言必备的标准。

4.2　清洁工具大 PK：讨论常用的垃圾回收算法

上一节讨论了垃圾回收的基本概念，读者应该已经知道什么是垃圾回收和为什么要进行自动化的垃圾回收了。本节将进一步讨论实现垃圾回收的方法，主要内容是理解 Java 垃圾回收机制的理论基础，将主要讨论：引用计数法、标记压缩法、标记清除法、复制算法和分代、分区的思想。

4.2.1 引用计数法（Reference Counting）

引用计数法是最经典也是最古老的一种垃圾收集方法，在微软的 COM 组件技术、Adobe 的 ActionScript3 中，都可以找到引用计数器的身影。

引用计数器的实现很简单，对于一个对象 A，只要有任何一个对象引用了 A，则 A 的引用计数器就加 1，当引用失效时，引用计数器就减 1。只要对象 A 的引用计数器的值为 0，则对象 A 就不可能再被使用。

引用计数器的实现也非常简单，只需要为每个对象配备一个整型的计数器即可。但是，引用计数器有两个非常严重的问题：

（1）无法处理循环引用的情况。因此，在 Java 的垃圾回收器中，没有使用这种算法。

（2）引用计算器要求在每次因引用产生和消除的时候，需要伴随一个加法操作和减法操作，对系统性能会有一定的影响。

一个简单的循环引用问题描述如下：有对象 A 和对象 B，对象 A 中含有对象 B 的引用，对象 B 中含有对象 A 的引用。此时，对象 A 和 B 的引用计数器都不为 0。但是，在系统中，却不存在任何第 3 个对象引用了 A 或 B。也就是说，A 和 B 是应该被回收的垃圾对象，但由于垃圾对象间相互引用，从而使垃圾回收器无法识别，引起内存泄漏。

如图 4.1 所示，不可达的对象出现循环引用，它的引用计数器均不为 0。

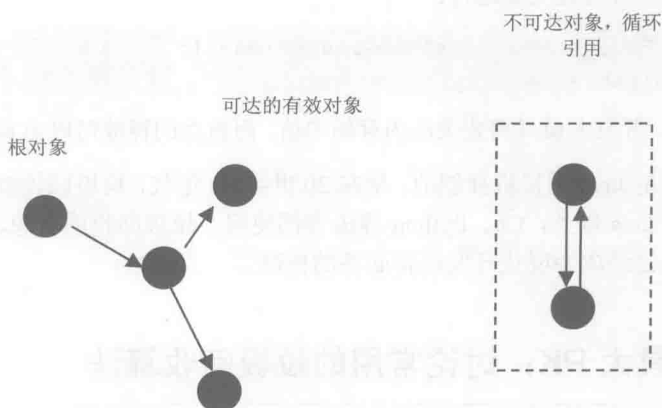


图 4.1 循环引用示意图

注意：由于单纯的引用计数算法隐含着循环引用以及性能问题，Java 虚拟机并未选择此算法作为垃圾回收算法。

【名词解释】

- **可达对象**：指通过根对象进行引用搜索，最终可以达到的对象。
- **不可达对象**：通过根对象进行引用搜索，最终没有被引用到的对象。

4.2.2　标记清除法（Mark-Sweep）

标记清除算法是现代垃圾回收算法的思想基础。标记清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。标记清除算法可能产生的最大问题是空间碎片。

如图 4.2 所示，使用标记清除算法对一块连续的内存空间进行回收。从根节点开始（这里显示了 2 个根），所有的有引用关系的对象均被标记为存活对象（箭头表示引用）。从根节点起，不可达的对象均为垃圾对象。在标记操作完成后，系统回收所有不可达的空间。

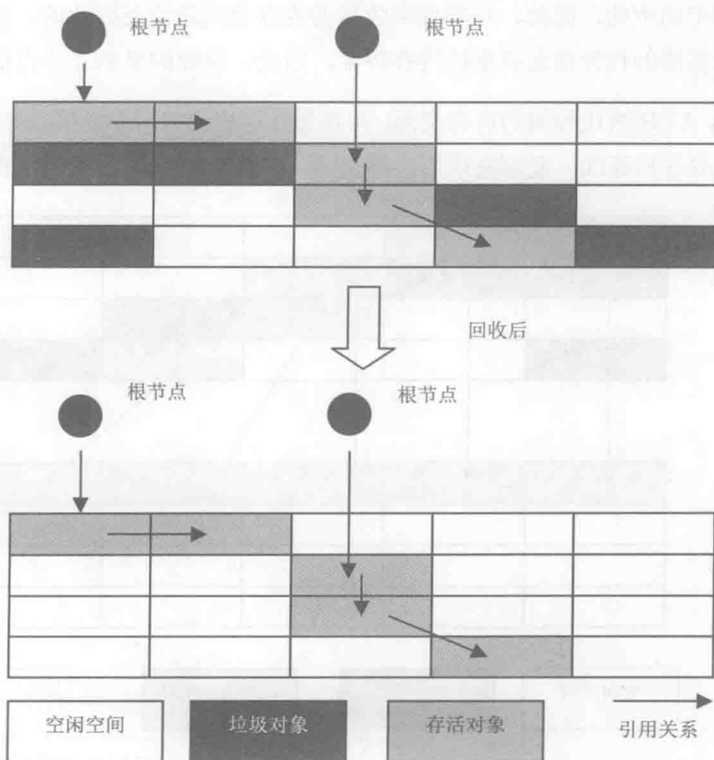


图 4.2　标记清除算法工作图

如图 4.2 所示，回收后的空间是不连续的。在对象的堆空间分配过程中，尤其是大对象的内存分配，不连续内存空间的工作效率要低于连续的空间。因此，这也是该算法的最大缺点。

注意： 标记清除算法先通过根节点标记所有可达对象，然后清除所有不可达对象，完成垃圾回收。

4.2.3 复制算法（Copying）

复制算法的核心思想是：将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。

如果系统中的垃圾对象很多，复制算法需要复制的存活对象数量就会相对较少。因此，在真正需要垃圾回收的时刻，复制算法的效率是很高的。又由于对象是在垃圾回收过程中，统一被复制到新的内存空间中的，因此，可确保回收后的内存空间是没有碎片的。虽然有以上两大优点，但是，复制算法的代价却是将系统内存折半，因此，单纯的复制算法也很难让人接受。

如图 4.3 所示，A、B 两块相同的内存空间，A 在进行垃圾回收时，将存活对象复制到 B 中，B 中的空间在复制后保持连续。复制完成后，清空 A。并将空间 B 设置为当前使用空间。

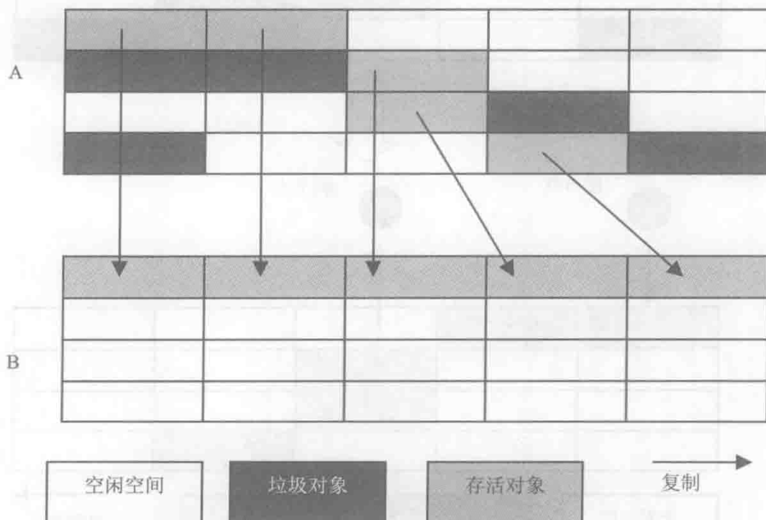


图 4.3　复制算法工作示意图

在 Java 的新生代串行垃圾回收器中，使用了复制算法的思想。新生代分为 eden 空间、from 空间和 to 空间 3 个部分。其中 from 和 to 空间可以视为用于复制的两块大小相同、地位相等、且可进行角色互换的空间块。from 和 to 空间也称为 survivor 空间，即幸存者空间，用于存放未被回收的对象。如图 4.4 所示。

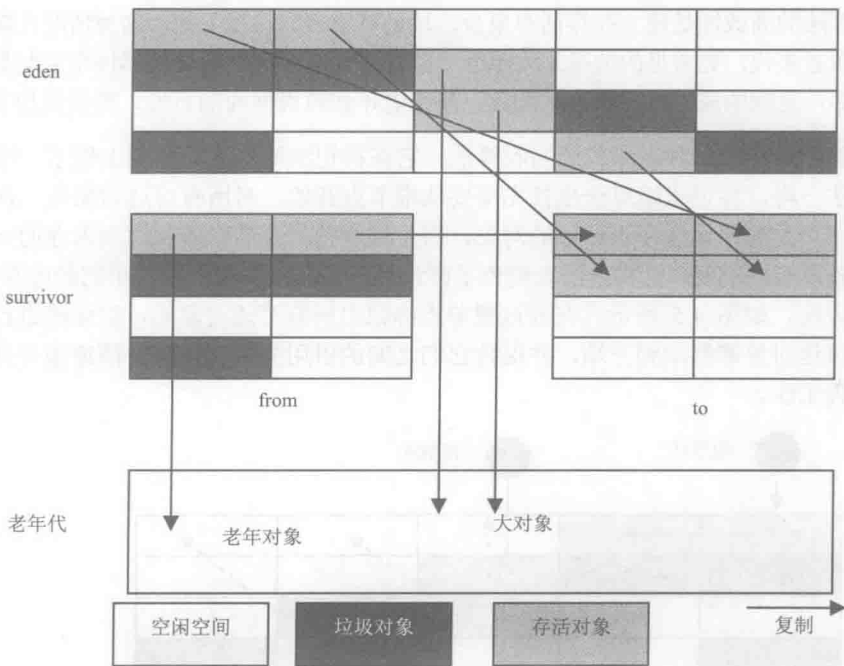


图 4.4　改进的复制算法

【名词解释】

- **新生代**：存放年轻对象的堆空间。年轻对象指刚刚创建的，或者经历垃圾回收次数不多的对象。
- **老年代**：存放老年对象的堆空间。老年对象指经历过多次垃圾回收依然存活的对象。

在垃圾回收时，eden 空间中的存活对象会被复制到未使用的 survivor 空间中（假设是 to），正在使用的 survivor 空间（假设是 from）中的年轻对象也会被复制到 to 空间中（大对象，或者老年对象会直接进入老年代，如果 to 空间已满，则对象也会直接进入老年代）。此时，eden 空间和 from 空间中的剩余对象就是垃圾对象，可以直接清空，to 空间则存放此次回收后的存活对象。这种改进的复制算法，既保证了空间的连续性，又避免了大量的内存空间浪费。如图 4.4 所示，显示了复制算法的实际回收过程。当所有存活对象都复制到 survivor 区后（图中为 to），简单地清空 eden 区和备用的 survivor 区（图中为 from）即可。

注意：复制算法比较适用于新生代。因为在新生代，垃圾对象通常会多于存活对象。复制算法的效果会比较好。

4.2.4 标记压缩法（Mark-Compact）

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。这种情况在新生代经常发生，但是在老年代，更常见的情况是大部分对象都是存活对象。如果依然使用复制算法，由于存活对象较多，复制的成本也将很高。因此，基于老年代垃圾回收的特性，需要使用其他的算法。

标记压缩算法是一种老年代的回收算法。它在标记清除算法的基础上做了一些优化。和标记清除算法一样，标记压缩算法也首先需要从根节点开始，对所有可达对象做一次标记。但之后，它并不只是简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比较高。如图 4.5 所示，在通过根节点标记出所有可达对象后，沿虚线进行对象移动，将所有的可达对象都移动到一端，并保持它们之间的引用关系，最后，清理边界外的空间，即可完成回收工作。

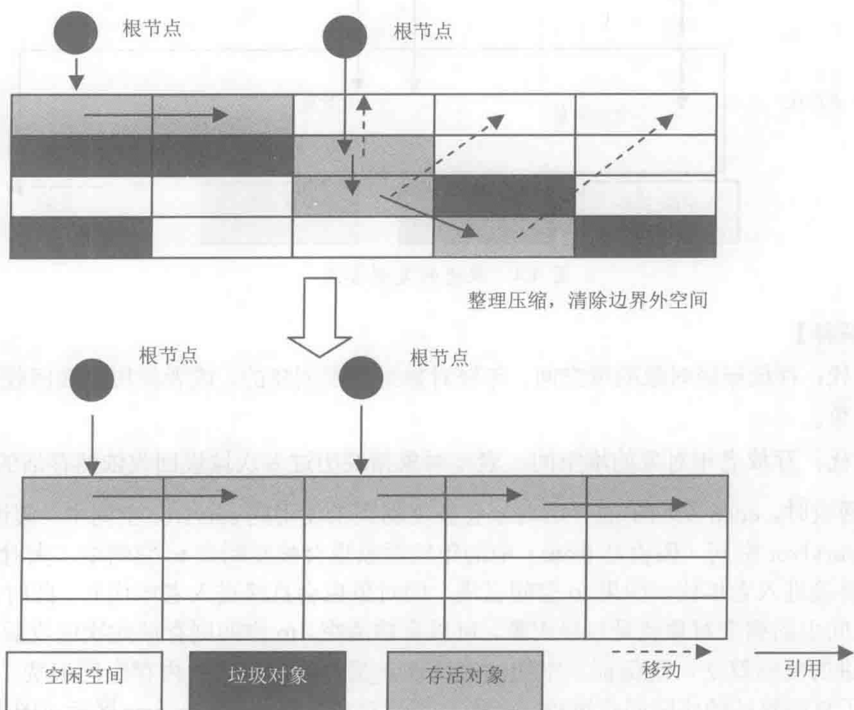


图 4.5 标记压缩算法工作示意图

标记压缩算法的最终效果等同于标记清除算法执行完成后，再进行一次内存碎片整理，因此，也可以把它称为标记清除压缩（MarkSweepCompact）算法。

4.2.5　分代算法（Generational Collecting）

前文中介绍了复制、标记清除、标记压缩等垃圾回收算法。在所有这些算法中，并没有一种算法可以完全替代其他算法，它们都具有自己独特的优势和特点。因此，根据垃圾回收对象的特性，使用合适的算法回收，才是明智的选择。

分代算法就是基于这种思想，它将内存区间根据对象的特点分成几块，根据每块内存区间的特点，使用不同的回收算法，以提高垃圾回收的效率。

一般来说，Java 虚拟机会将所有的新建对象都放入称为新生代的内存区域，新生代的特点是对象朝生夕灭，大约 90%的新建对象会被很快回收，因此，新生代比较适合使用复制算法。当一个对象经过几次回收后依然存活，对象就会被放入称为老年代的内存空间。在老年代中，几乎所有的对象都是经过几次垃圾回收后依然得以存活的。因此，可以认为这些对象在一段时期内，甚至在应用程序的整个生命周期中，将是常驻内存的。

在极端情况下，老年代对象的存活率可以达到 100%。如果依然使用复制算法回收老年代，将需要复制大量对象。再加上老年代的回收性价比也要低于新生代，因此这种做法是不可取的。根据分代的思想，可以对老年代的回收使用与新生代不同的标记压缩或标记清除算法，以提高垃圾回收效率。如图 4.6 所示，显示了这种分代回收的思想。

注意：分代的思想被现有的虚拟机广泛使用。几乎所有的垃圾回收器都区分新生代和老年代。

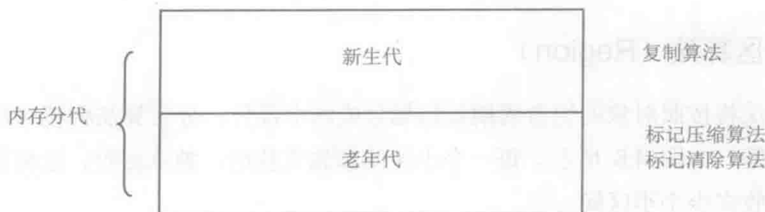


图 4.6　分代回收的思想

对于新生代和老年代来说，通常，新生代回收的频率很高，但是每次回收的耗时都很短，而老年代回收的频率比较低，但是会消耗更多的时间。为了支持高频率的新生代回收，虚拟机可能使用一种叫作卡表（Card Table）的数据结构。卡表为一个比特位集合，每一个比特位可以

用来表示老年代的某一区域中的所有对象是否持有新生代对象的引用。这样在新生代 GC 时，可以不用花大量时间扫描所有老年代对象，来确定每一个对象的引用关系，而可以先扫描卡表，只有当卡表的标记位为 1 时，才需要扫描给定区域的老年代对象，而卡表位为 0 的所在区域的老年代对象，一定不含有新生代对象的引用。如图 4.7 所示，卡表中每一位表示老年代 4KB 的空间，卡表记录为 0 的老年代区域没有任何对象指向新生代，只有卡表位为 1 的区域才有对象包含新生代引用，因此在新生代 GC 时，只需要扫描卡表位为 1 所在的老年代空间。使用这种方式，可以大大加快新生代的回收速度。

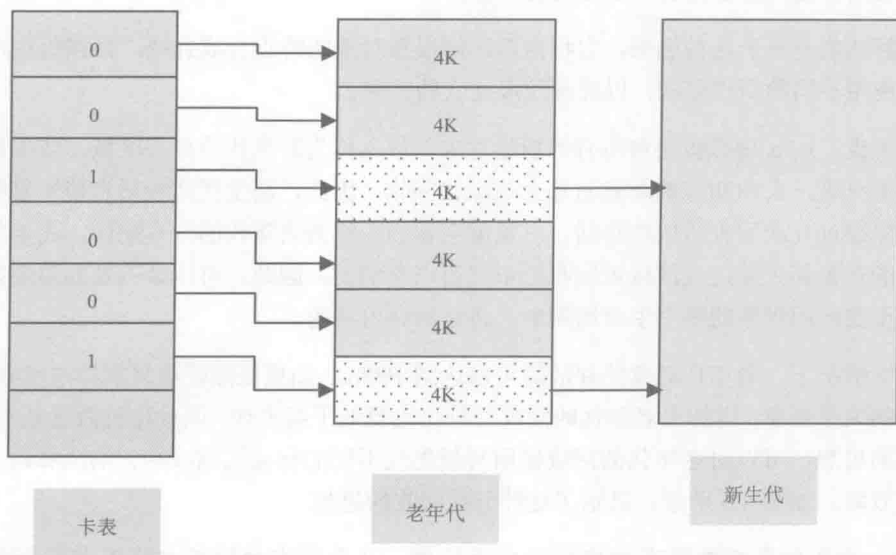


图 4.7　新生代 GC 根据卡表，只需扫描部分老年代

4.2.6　分区算法（Region）

分代算法将按照对象的生命周期长短划分成两个部分，分区算法将整个堆空间划分成连续的不同小区间，如图 4.8 所示。每一个小区间都独立使用，独立回收。这种算法的好处是可以控制一次回收多少个小区间。

一般来说，在相同条件下，堆空间越大，一次 GC 时所需要的时间就越长，从而产生的停顿也越长，有关 GC 产生的停顿，请参见 4.4 节。为了更好地控制 GC 产生的停顿时间，将一块大的内存区域分割成多个小块，根据目标的停顿时间，每次合理地回收若干个小区间，而不是整个堆空间，从而减少一次 GC 所产生的停顿。

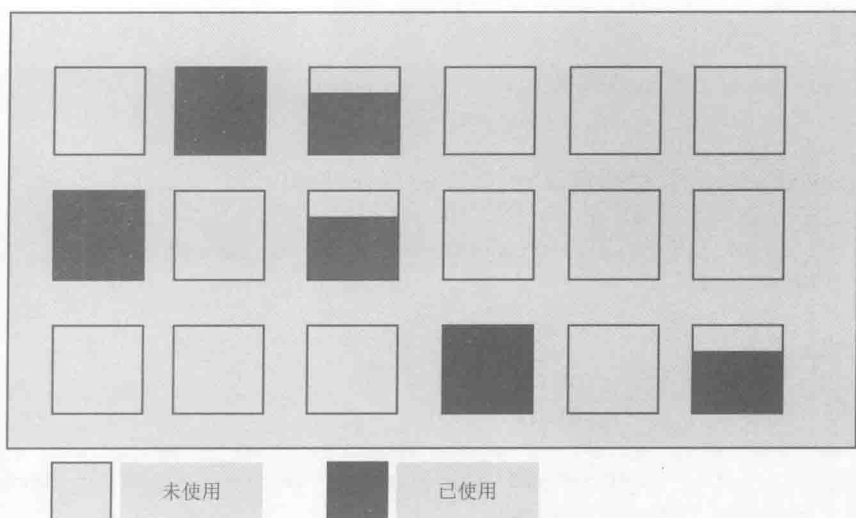


图 4.8　分区算法示意图

4.3　谁才是真正的垃圾：判断可触及性

垃圾回收的基本思想是考察每一个对象的可触及性，即从根节点开始是否可以访问到这个对象，如果可以，则说明当前对象正在被使用，如果从所有的根节点都无法访问到某个对象，说明对象已经不再使用了，一般来说，此对象需要被回收。但事实上，一个无法触及的对象有可能在某一个条件下“复活”自己，如果这样，那么对它的回收就是不合理的，为此，需要给出一个对象可触及性状态的定义，并规定在什么状态下，才可以安全地回收对象。

简单来说，可触及性可以包含以下 3 种状态。

- **可触及的**：从根节点开始，可以到达这个对象。
- **可复活的**：对象的所有引用都被释放，但是对象有可能在 finalize()函数中复活。
- **不可触及的**：对象的 finalize()函数被调用，并且没有复活，那么就会进入不可触及状态，不可触及的对象不可能被复活，因为 finalize()函数只会被调用一次。

以上 3 种状态中，只有在对象不可触及时才可以被回收。

4.3.1　对象的复活

【示例 4-1】前文中已经提到，对象有可能在 finalize()函数中复活自己，这里给出一个例子，

演示这种情况。

```
public class CanReliveObj {
  public static CanReliveObj obj;
  @Override
  protected void finalize() throws Throwable {
      super.finalize();
      System.out.println("CanReliveObj finalize called");
      obj=this;
  }
  @Override
  public String toString(){
      return "I am CanReliveObj";
  }
  public static void main(String[] args) throws InterruptedException{
      obj=new CanReliveObj();
      obj=null;
      System.gc();
      Thread.sleep(1000);
      if(obj==null){
          System.out.println("obj 是 null");
      }else{
          System.out.println("obj 可用");
      }
      System.out.println("第 2 次 gc");
      obj=null;
      System.gc();
      Thread.sleep(1000);
      if(obj==null){
          System.out.println("obj 是 null");
      }else{
          System.out.println("obj 可用");
      }
  }
}
```

运行以上代码，输出如下：

```
CanReliveObj finalize called
obj 可用
第 2 次 gc
obj 是 null
```

可以看到，在代码第 15 行将 obj 设置为 null 后，进行 GC，结果发现 obj 对象被复活了。第 24 行，再次释放对象引用并运行 GC，对象才真正地被回收。这是因为第一次 GC 时，在 finalize()函数调用之前，虽然系统中的引用已经被清除，但是作为实例方法 finalize()，对象的 this 引用依然会被传入方法内部，如果引用外泄，对象就会复活，此时，对象又变为可触及状态。而 finalize()函数只会被调用一次，因此，第 2 次清除对象时，对象就再无机会复活，因此就会被回收。

注意：finalize()函数是一个非常糟糕的模式，再次不推荐读者使用 finalize()函数释放资源。

第一，因为 finalize()函数有可能发生引用外泄，在无意中复活对象；

第二，由于 finalize()是被系统调用的，调用时间是不明确的，因此不是一个好的资源释放方案，推荐在 try-catch-finally 语句中进行资源的释放。

4.3.2 引用和可触及性的强度

在 Java 中提供了 4 个级别的引用：强引用、软引用、弱引用和虚引用。除强引用外，其他 3 种引用均可以在 java.lang.ref 包中找到它们的身影。如图 4.9 所示，显示了这 3 种引用类型对应的类，开发人员可以在应用程序中直接使用它们。其中 FinalReference 意味“最终”引用，它用以实现对象的 finalize()方法（更多相关内容请参见 5.5.6 节）。

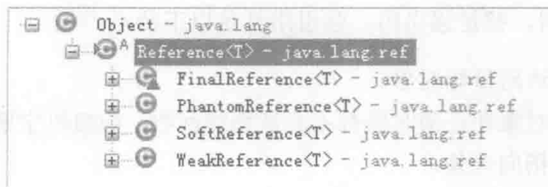


图 4.9 引用类型

强引用就是程序中一般使用的引用类型，强引用的对象是可触及的，不会被回收。相对的，软引用、弱引用和虚引用的对象是软可触及、弱可触及和虚可触及的，在一定条件下，都是可以被回收的。

【示例 4-2】下面是一个强引用的例子：

```
StringBuffer str = new StringBuffer ("Hello world");
```

假设以上代码是在函数体内运行的，那么局部变量 str 将被分配在栈上，而对象 StringBuffer 实例被分配在堆上。局部变量 str 指向 StringBuffer 实例所在堆空间，通过 str 可以操作该实例，那么 str 就是 StringBuffer 实例的强引用，如图 4.10 所示。

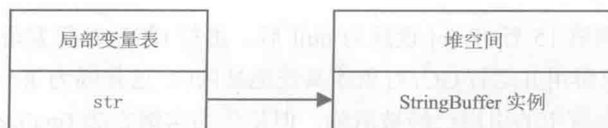


图 4.10　强引用示例 1

此时，如果再运行一个赋值语句：

```
StringBuffer str1=str;
```

那么，str 所指向的对象也将被 str1 所指向，同时在局部变量表上会分配空间存放 str1 变量，如图 4.11 所示。此时，该 StringBuffer 实例就有两个引用。对引用的“==”操作用于表示两操作数所指向的堆空间地址是否相同，不表示两操作数所指向的对象是否相等。

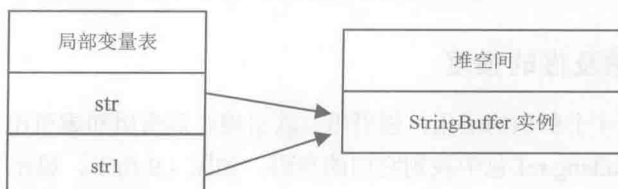


图 4.11　强引用示例 2

如本例中的两个引用，都是强引用，强引用具备以下特点：

- 强引用可以直接访问目标对象。
- 强引用所指向的对象在任何时候都不会被系统回收，虚拟机宁愿抛出 OOM 异常，也不会回收强引用所指向对象。
- 强引用可能导致内存泄漏。

4.3.3 软引用——可被回收的引用

软引用是比强引用弱一点的引用类型。一个对象只持有软引用，那么当堆空间不足时，就会被回收。软引用使用 java.lang.ref.SoftReference 类实现。

【示例 4-3】下面示例演示了软引用会在系统堆内存不足时被回收。

```
public class SoftRef {
    public static class User{
        public User(int id,String name){
            this.id=id;
            this.name=name;
```

```
06          }
07          public int id;
08          public String name;
09
10          @Override
11          public String toString(){
12              return "[id="+String.valueOf(id)+",name="+name+"]";
13          }
14      }
15      public static void main(String[] args) {
16          User u=new User(1,"geym");
17          SoftReference<User> userSoftRef = new SoftReference<User>(u);
18          u=null;
19
20          System.out.println(userSoftRef.get());
21          System.gc();
22          System.out.println("After GC:");
23          System.out.println(userSoftRef.get());
24
25          byte[] b=new byte[1024*925*7];
26          System.gc();
27          System.out.println(userSoftRef.get());
28      }
29  }
```

上述代码申明了一个 User 类，在代码第 16 行，建立了 User 类的实例，这里的 u 变量为强引用。代码第 17 行，通过强引用 u，建立软引用。代码第 18 行，去除强引用。代码第 20 行从软引用中重新获得强引用对象。第 21 行进行一次垃圾回收，第 23 行，在垃圾回收之后，在此获得软引用中的对象。在第 25 行，分配一块较大的内存，让系统认为内存资源紧张，在 26 行进行一次 GC（实际上，这个是多余的，因为在分配大数据时，系统会自动进行 GC，这里只是为了更清楚地说明问题），第 27 行再次从软引用中获取数据。

使用参数-Xmx10m 运行上述代码，得到：

```
[id=1,name=geym]   （第一次从软引用中获取数据）
After GC:
[id=1,name=geym]   （GC 没有清除软引用）
null               （由于内存紧张，软引用被清除）
```

因此，从该示例中可以得到结论：GC 未必会回收软引用的对象，但是，当内存资源紧张时，软引用对象会被回收，所以软引用对象不会引起内存溢出。

【示例 4-4】每一个软引用都可以附带一个引用队列，当对象的可达性状态发生改变时（由可达变为不可达），软引用对象就会进入引用队列。通过这个引用队列，可以跟踪对象的回收情况。请看下面的代码：

```
public class SoftRefQ {
    public static class User{
        public User(int id,String name){
            this.id=id;
            this.name=name;
        }
        //省略部分代码，请参考 SoftRef.User 类
    }

    static ReferenceQueue<User> softQueue=null;
    public static class CheckRefQueue extends Thread{
        @Override
        public void run(){
            while(true){
                if(softQueue!=null){
                    UserSoftReference obj=null;
                        try {
                            obj = (UserSoftReference) softQueue.remove();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                        if(obj!=null)
                        System.out.println("user id "+obj.uid+" is delete");
                }
            }
        }
    }

    public static class UserSoftReference extends SoftReference<User>{
        int uid;
        public UserSoftReference(User referent, ReferenceQueue<? super User> q) {
            super(referent, q);
            uid=referent.id;
        }
    }
```

```
36
37     public static void main(String[] args) throws InterruptedException {
38         Thread t=new CheckRefQueue();
39         t.setDaemon(true);
40         t.start();
41         User u=new User(1,"geym");
42         softQueue = new ReferenceQueue<User>();
43         UserSoftReference userSoftRef = new UserSoftReference(u,softQueue);
44         u=null;
45         System.out.println(userSoftRef.get());
46         System.gc();
47         //内存足够，不会被回收
48         System.out.println("After GC:");
49         System.out.println(userSoftRef.get());
50
51         System.out.println("try to create byte array and GC");
52         byte[] b=new byte[1024*925*7];
53         System.gc();
54         System.out.println(userSoftRef.get());
55
56         Thread.sleep(1000);
57     }
58 }
```

上述代码值得注意的地方有两个：

（1）第 29~35 行实现了一个自定义的软引用类，扩展软引用的目的是记录 User.uid，后续在引用队列中，就可以通过这个 uid 字段知道哪个 User 实例被回收了。

（2）第 2 个值得注意的地方是代码第 43 行，在创建软引用时，指定了一个软引用队列，当给定的对象实例被回收时，就会被加入这个引用队列，通过访问该队列可以跟踪对象的回收情况，代码 14~24 行就是跟踪引用队列，打印对象的回收情况。

使用参数-Xmx10m 运行上述代码可以得到：

```
[id=1,name=geym]                  （第一次从软引用获得对象）
After GC:
[id=1,name=geym]                  （GC 后，软引用对象没有回收）
try to create byte array and GC      （创建大数组，耗尽内存）
user id 1 is delete                 （引用队列探测到对象被删除）
null                             （对象已被回收，无法再通过软引用获取对象）
```

4.3.4 弱引用——发现即回收

弱引用是一种比软引用较弱的引用类型。在系统 GC 时，只要发现弱引用，不管系统堆空间使用情况如何，都会将对象进行回收。但是，由于垃圾回收器的线程通常优先级很低，因此，并不一定能很快地发现持有弱引用的对象。在这种情况下，弱引用对象可以存在较长的时间。一旦一个弱引用对象被垃圾回收器回收，便会加入到一个注册的引用队列中（这一点和软引用很像）。弱引用使用 java.lang.ref.WeakReference 类实现。

【示例 4-5】下面的例子显示了弱引用的特点。

```
01 public class WeakRef {
02     public static class User{
03         public User(int id,String name){
04             this.id=id;
05             this.name=name;
06         }
07        //省略部分代码，请参考 SoftRef.User 类
08     }
09     public static void main(String[] args) {
10         User u=new User(1,"geym");
11         WeakReference<User> userWeakRef = new WeakReference<User>(u);
12         u=null;
13         System.out.println(userWeakRef.get());
14         System.gc();
15         //不管当前内存空间足够与否，都会回收它的内存
16         System.out.println("After GC:");
17         System.out.println(userWeakRef.get());
18     }
19 }
```

上述代码第 11 行，构造了弱引用。代码 12 行，去除了强引用。代码第 13 行从弱引用中重新获取对象。第 14 行进行 GC，第 17 行重新尝试从弱引用中获取对象。

运行上述代码，输出为：

```
[id=1,name=geym]
After GC:
Null
```

可以看到，在 GC 之后，弱引用对象立即被清除。

弱引用和软引用一样，在构造弱引用时，也可以指定一个引用队列，当弱引用对象被回收

时，就会加入指定的引用队列，通过这个队列可以跟踪对象的回收情况。读者可以参考 4.3.3 节中的介绍自行实现，在此不再赘述。

注意： 软引用、弱引用都非常适合来保存那些可有可无的缓存数据。如果这么做，当系统内存不足时，这些缓存数据会被回收，不会导致内存溢出。而当内存资源充足时，这些缓存数据又可以存在相当长的时间，从而起到加速系统的作用。

4.3.5　虚引用——对象回收跟踪

虚引用是所有引用类型中最弱的一个。一个持有虚引用的对象，和没有引用几乎是一样的，随时都可能被垃圾回收器回收。当试图通过虚引用的 get()方法取得强引用时，总是会失败。并且，虚引用必须和引用队列一起使用，它的作用在于跟踪垃圾回收过程。

当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象后，将这个虚引用加入引用队列，以通知应用程序对象的回收情况。

【示例 4-6】下面给出一个示例，使用虚引用跟踪一个可复活对象的回收。

```
public class TraceCanReliveObj {
 public static TraceCanReliveObj obj;
  static ReferenceQueue<TraceCanReliveObj> phantomQueue=null;
     public static class CheckRefQueue extends Thread{
         @Override
         public void run(){
             while(true){
                 if(phantomQueue!=null){
                    PhantomReference<TraceCanReliveObj> objt=null;
                         try {
                objt = (PhantomReference<TraceCanReliveObj>)phantomQueue.
remove();
                              } catch (InterruptedException e) {
                                 e.printStackTrace();
                              }
                              if(objt!=null){
                                System.out.println("TraceCanReliveObj is delete
by GC");
                              }
                 }
             }
         }
```

```
    }

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("CanReliveObj finalize called");
        obj=this;
    }
    @Override
    public String toString(){
        return "I am CanReliveObj";
    }
    public static void main(String[] args) throws InterruptedException{
        Thread t=new CheckRefQueue();
        t.setDaemon(true);
        t.start();

        phantomQueue = new ReferenceQueue<TraceCanReliveObj>();
        obj=new TraceCanReliveObj();
        PhantomReference<TraceCanReliveObj> phantomRef = new PhantomReference<TraceCanReliveObj>(obj,phantomQueue);

        obj=null;
        System.gc();
        Thread.sleep(1000);
        if(obj==null){
            System.out.println("obj 是 null");
        }else{
            System.out.println("obj 可用");
        }
        System.out.println("第 2 次 gc");
        obj=null;
        System.gc();
        Thread.sleep(1000);
        if(obj==null){
            System.out.println("obj 是 null");
        }else{
            System.out.println("obj 可用");
        }
    }
}
```

上述代码中 TraceCanReliveObj 对象是一个在 finalize()函数中可复活的对象。在代码第 40 行，构造了 TraceCanReliveObj 对象的虚引用，并指定了引用队列。第 42 行将强引用去除。第 43 行第一次进行 GC，由于对象可复活，GC 无法回收该对象。接着，在第 52 行进行第 2 次 GC，由于 finalize()只会被调用一次，因此第 2 次 GC 会回收对象，同时其引用队列应该也会捕获到对象的回收。

执行上述代码，得到如下输出：

```
CanReliveObj finalize called          （对象复活）
obj 可用
第 2 次 gc                     （第 2 次，对象无法复活）
TraceCanReliveObj is delete by GC    （引用队列捕获到对象被回收）
obj 是 null
```

由于虚引用可以跟踪对象的回收时间，因此，也可以将一些资源释放操作放置在虚引用中执行和记录。

4.4 垃圾回收时的停顿现象：Stop-The-World 案例实战

垃圾回收器的任务是识别和回收垃圾对象进行内存清理。为了让垃圾回收器可以正常且高效地执行，大部分情况下，会要求系统进入一个停顿的状态。停顿的目的是终止所有应用线程的执行，只有这样，系统中才不会有新的垃圾产生，同时停顿保证了系统状态在某一个瞬间的一致性，也有益于垃圾回收器更好地标记垃圾对象。因此，在垃圾回收时，都会产生应用程序的停顿。停顿产生时，整个应用程序会被卡死，没有任何响应，因此这个停顿也叫做“Stop-The-World”（STW）。

注意：本节中涉及的一些虚拟机参数会在第 5 章中介绍，读者可以翻阅参考。

【示例 4-7】下面这个示例演示了这种停顿的情况。

```
public class StopWorldTest {
 public static class MyThread extends Thread{
     HashMap map=new HashMap();
     @Override
     public void run(){
         try{
             while(true){
                 if(map.size()*512/1024/1024>=900){
                     map.clear();
```

```
10                     System.out.println("clean map");
11                 }
12                 byte[] b1;
13                 for(int i=0;i<100;i++){
14                     b1=new byte[512];
15                     map.put(System.nanoTime(), b1);
16                 }
17                 Thread.sleep(1);
18             }
19         }catch(Exception e){
20 
21         }
22     }
23 }
24 public static class PrintThread extends Thread{
25     public static final long starttime=System.currentTimeMillis();
26     @Override
27     public void run(){
28         try{
29             while(true){
30                 long t=System.currentTimeMillis()-starttime;
31                 System.out.println(t/1000+"."+t%1000);
32                 Thread.sleep(100);
33             }
34         }catch(Exception e){
35 
36         }
37     }
38 }
39 public static void main(String args[]){
40     MyThread t=new MyThread();
41     PrintThread p=new PrintThread();
42     t.start();
43     p.start();
44 }
45 }
```

上述代码中，开启两个线程，PrintThread 负责每 0.1 秒在控制台上进行一次时间戳的输出，MyThread 则不停地消耗内存资源，以引起 GC。代码第 8 行，在内存消耗大于 900MB 时，清空内存，防止内存溢出。

使用参数：

```
-Xmx1g -Xms1g -Xmn512k -XX:+UseSerialGC -Xloggc:gc.log -XX:+PrintGCDetails
```

运行上述代码，此参数设置了 1GB 的堆空间，并基本废弃了新生代（只保留 512KB）。运行程序得到的部分输出如下：

```
24.618
24.718
24.818
24.918
25.120
25.956
26.56
26.755
27.363
28.179
clean map
```

注意加粗部分，原本应该每 0.1 秒进行的输出在这几处有着明显的时间间隔。找到对应的 GC 日志输出如下：

```
25.286: [GC25.286: [DefNew: 448K->448K(448K), 0.0000136 secs]25.286: [Tenured: 1047987K->1040135K(1048064K), 0.7662287 secs] 1048435K->1040135K(1048512K), [Perm : 4677K->4677K(12288K)], 0.7663218 secs] [Times: user=0.76 sys=0.00, real=0.77 secs]
26.248: [GC26.248: [DefNew: 447K->447K(448K), 0.0000132 secs]26.248: [Tenured: 1047947K->1048063K(1048064K), 0.6036506 secs] 1048395K->1048359K(1048512K), [Perm: 4677K->4677K(12288K)], 0.6037413 secs] [Times: user=0.61 sys=0.00, real=0.60 secs]
26.854: [Full GC26.854: [Tenured: 1048063K->1048063K(1048064K), 0.6053799 secs] 1048511K->1048485K(1048512K), [Perm : 4677K->4677K(12288K)], 0.6054616 secs] [Times: user=0.61 sys=0.00, real=0.61 secs]
27.460: [Full GC27.460: [Tenured: 1048063K->1039889K(1048064K), 0.8147534 secs] 1048511K->1039889K(1048512K), [Perm : 4677K->4619K(12288K)], 0.8148368 secs] [Times: user=0.81 sys=0.00, real=0.82 secs]
```

注意加粗部分的 GC 时间戳和实际花费的时间，不难发现，在程序打印的 25.120~25.956 秒处，发生了大约 0.8 秒左右的停顿，对应于 GC 日志中 25.286 秒的 Full GC。程序打印 25.956~26.56 秒处，发生大约 0.6 秒的停顿，对于 GC 日志 26.248 秒处的 0.6 秒停顿。应用输出 26.755~27.363 秒处约 0.6 秒的停顿，对应于 GC 日志 26.854 秒处的 0.61 秒的停顿。最后，程序 27.363~28.179 秒处约 0.8 秒左右的停顿，则来自于 GC 日志 27.460 秒处的 0.82 秒的停顿。由此可见，每一次

应用程序的意外停顿都可以在 GC 日志中找到对应的线索给予解释。这也间接证明了 GC 对于应用程序的影响。

笔者使用 Visual GC 观察上述程序的运行过程，结果如图 4.12 所示，可以看到老年代 GC 共进行 5 次，合计耗时 2.9 秒，平均一次约 0.6 秒左右，而新生代 GC 合计进行 3895 次，合计耗时 5.1 秒多，GC 总耗时约 8 秒。

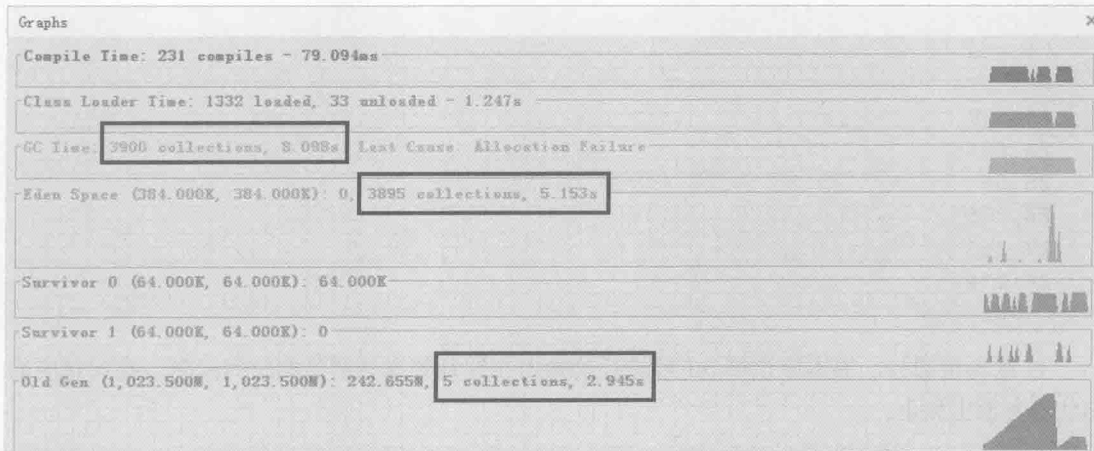


图 4.12　Visual GC 观察 GC 过程 1

从这个例子中可以看到，新生代 GC 进行比较频繁，但每一次 GC 耗时较短，老年代 GC 发生次数较少，但每一次所消耗的时间较长。这种现象和虚拟机参考设置有关。下面通过修改虚拟机参数改变这种现象。

使用下面的虚拟机参数执行上述代码：

```
-Xmx1g -Xms1g -Xmn900m -XX:SurvivorRatio=1 -XX:+UseSerialGC -Xloggc:gc.log
-XX:+PrintGCDetails
```

此参数设置了一个较大的新生代（900MB），并将 from、to 区域和 eden 区域设置各 300MB。同时，修改上述代码第 8 行为（读者考虑一下为何需要做这个修改）：

```
if(map.size()*512/1024/1024>=550){
```

使用这种超大新生代的设置，会导致复制算法复制大量对象，也会在很大程度上延长 GC 时间。程序的部分输出如下：

```
10.251
10.351
```

```
11.409
11.511
11.611
```

可以看到，在 10.351~11.409 处，产生了大约 1 秒的停顿。翻阅 GC 日志，不难发现：

```
10.541: [GC10.541: [DefNew (promotion failed) : 559190K->548752K(614400K),
0.5119172 secs]11.053: [Tenured: 126976K->126976K(126976K), 0.4484432 secs]
559190K->489440K(741376K), [Perm : 4657K->4657K(12288K)], 0.9604906 secs] [Times:
user=0.91 sys=0.05, real=0.96 secs]
```

使用 Visual GC 观察这次行为，结果如图 4.13 所示。

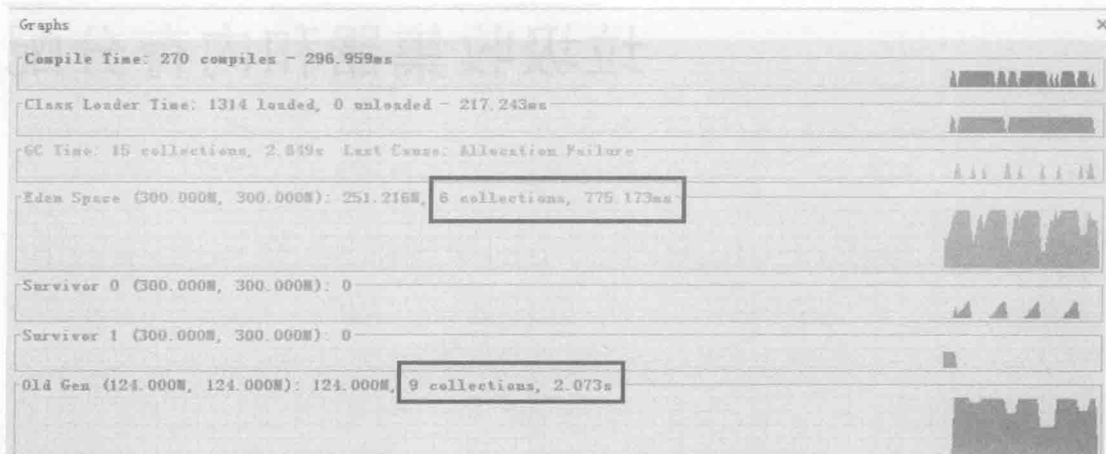


图 4.13　Visual GC 观察 GC 过程 2

可以看到，在增大新生代空间后，新生代 GC 次数明显减少，但是每次耗时增加，这里显示的 6 次新生代 GC 合计耗时 775ms。

4.5　小结

本章主要介绍了垃圾回收的理论基础，包括一些主要的垃圾回收算法思想。同时，也介绍了与垃圾回收相关的基本概念，如可触及性、可达对象、新生代、老年代等，并且给出了一个有关垃圾回收带来系统停顿的案例。希望读者能熟读这些内容，为将来了解 Java 的垃圾回收器打下一个良好的基础。

5

第 5 章 垃圾收集器和内存分配

在 Java 虚拟机中，垃圾回收器可不仅仅只有一种，什么情况下要使用哪一种，对性能又有什么样的影响，这都是我们必须要了解的。本章将具体介绍虚拟机中的垃圾回收器类型，以及它们的特点和使用方法，并进一步探讨有关对象在内存中的分配和回收的问题。

本章涉及的主要知识点有：

- Java 虚拟机支持的垃圾收集器种类。
- 串行垃圾回收器。
- 并行垃圾回收器。
- CMS 回收器。
- G1 回收器。
- 有关对象分配的一些细节问题。

5.1　一心一意一件事：串行回收器

串行回收器是指使用单线程进行垃圾回收的回收器。每次回收时，串行回收器只有一个工作线程，对于并行能力较弱的计算机来说，串行回收器的专注性和独占性往往有更好的性能表现。串行回收器可以在新生代和老年代使用，根据作用于不同的堆空间，分为新生代串行回收器和老年代串行回收器。

5.1.1　新生代串行回收器

串行收集器是所有垃圾回收器中最古老的一种，也是 JDK 中最基本的垃圾回收器之一。串行回收器主要有两个特点：

第一，它仅仅使用单线程进行垃圾回收。

第二，它是独占式的垃圾回收。

在串行收集器进行垃圾回收时，Java 应用程序中的线程都需要暂停，等待垃圾回收的完成。如图 5.1 所示，在串行回收器运行时，应用程序中的所有线程都停止工作，进行等待。这种现象称之为“Stop-The-World”。它将造成非常糟糕的用户体验，在实时性要求较高的应用场景中，这种现象往往是不能被接受的。

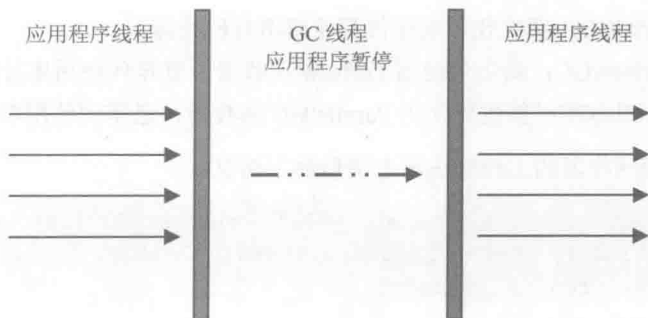


图 5.1　串行回收示意图

虽然如此，串行回收器却是一个成熟且经过长时间生产环境考验的极为高效的收集器。新生代串行处理器使用复制算法，实现相对简单、逻辑处理特别高效、且没有线程切换的开销。在诸如单 CPU 处理器等硬件平台不是特别优越的场合，它的性能表现可以超过并行回收器和并发回收器。

使用-XX:+UseSerialGC 参数可以指定使用新生代串行收集器和老年代串行收集器。当虚拟机在 Client 模式下运行时，它是默认的垃圾收集器。

一次新生代串行收集器的工作输出日志类似如下信息（使用-XX:+PrintGCDetails 开关）：

```
0.844: [GC 0.844: [DefNew: 17472K->2176K(19648K), 0.0188339 secs] 17472K->
2375K(63360K), 0.0189186 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
```

它显示了一次垃圾回收前的新生代内存占用量和垃圾回收后的新生代内存占用量，以及垃圾回收所消耗的时间。

注意： 串行垃圾回收器虽然古老，但是久经考验。在大多数情况下，其性能表现是相当不错的。

5.1.2 老年代串行回收器

老年代串行收集器使用的是标记压缩算法。和新生代串行收集器一样，它也是一个串行的、独占式的垃圾回收器。由于老年代垃圾回收通常会使用比新生代回收更长的时间，因此，在堆空间较大的应用程序中，一旦老年代串行收集器启动，应用程序很可能会因此停顿较长的时间。

虽然如此，作为老牌的垃圾回收器，老年代串行回收器可以和多种新生代回收器配合使用，同时它也可以作为 CMS 回收器的备用回收器。

若要启用老年代串行回收器，可以尝试使用以下参数。

- -XX:+UseSerialGC：新生代、老年代都使用串行回收器。
- -XX:+UseParNewGC：新生代使用 ParNew 回收器，老年代使用串行收集器。
- -XX:+UseParallelGC：新生代使用 ParallelGC 回收器，老年代使用串行收集器。

一次老年代串行回收器的工作输出日志类似如下信息：

```
8.259: [Full GC 8.259: [Tenured: 43711K->40302K(43712K), 0.2960477 secs]
63350K->40302K(63360K), [Perm : 17836K->17836K(32768K)], 0.2961554 secs] [Times:
user=0.28 sys=0.02, real=0.30 secs]
```

它显示了垃圾回收前老年代和永久区的内存占用量，以及垃圾回收后老年代和永久区的内存占用量。

5.2 人多力量大：并行回收器

并行回收器在串行回收器的基础上做了改进，它使用多个线程同时进行垃圾回收。对于并行能力强的计算机，可以有效缩短垃圾回收所需的实际时间。

5.2.1　新生代 ParNew 回收器

ParNew 回收器是一个工作在新生代的垃圾收集器。它只是简单地将串行回收器多线程化，它的回收策略、算法以及参数和新生代串行回收器一样。ParNew 回收器的工作示意图如图 5.2 所示。ParNew 回收器也是独占式的回收器，在收集过程中，应用程序会全部暂停。但由于并行回收器使用多线程进行垃圾回收，因此，在并发能力比较强的 CPU 上，它产生的停顿时间要短于串行回收器，而在单 CPU 或者并发能力较弱的系统中，并行回收器的效果不会比串行回收器好，由于多线程的压力，它的实际表现很可能比串行回收器差。

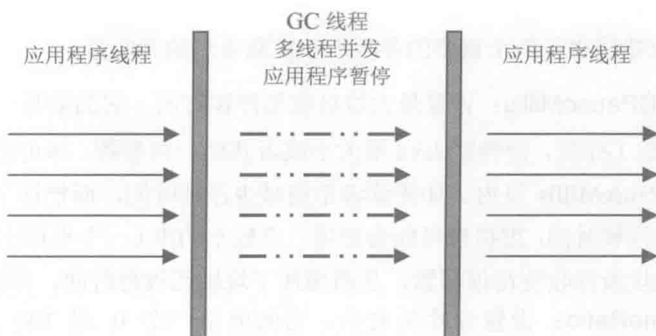


图 5.2　ParNew 回收器工作示意图

开启 ParNew 回收器可以使用以下参数。

- -XX:+UseParNewGC：新生代使用 ParNew 回收器，老年代使用串行回收器。
- -XX:+UseConcMarkSweepGC：新生代使用 ParNew 回收器，老年代使用 CMS。

ParNew 回收器工作时的线程数量可以使用-XX:ParallelGCThreads 参数指定。一般，最好与 CPU 数量相当，避免过多的线程数，影响垃圾收集性能。在默认情况下，当 CPU 数量小于 8 个时，ParallelGCThreads 的值等于 CPU 数量，当 CPU 数量大于 8 个时，ParallelGCThreads 的值等于 3+((5*CPU_Count)/8)。

一次 ParNew 回收器的日志输出信息如下：

```
0.834: [GC 0.834: [ParNew: 13184K->1600K(14784K), 0.0092203 secs] 13184K->
1921K(63936K), 0.0093401 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

可以看到，这个输出和新生代串行收集器几乎是一样的，只有回收器标识符不同。

5.2.2 新生代 ParallelGC 回收器

新生代 ParallelGC 回收器也是使用复制算法的收集器。从表面上看，它和 ParNew 回收器一样，都是多线程、独占式的收集器。但是， ParallelGC 回收器有一个重要的特点：它非常关注系统的吞吐量。

新生代 ParallelGC 回收器可以使用以下参数启用。

- –XX:+UseParallelGC：新生代使用 ParallelGC 回收器，老年代使用串行回收器。
- –XX:+UseParallelOldGC：新生代使用 ParallelGC 回收器，老年代使用 ParallelOldGC 回收器。

ParallelGC 回收器提供了两个重要的参数用于控制系统的吞吐量。

- –XX:MaxGCPauseMillis：设置最大垃圾收集停顿时间。它的值是一个大于 0 的整数。ParallelGC 在工作时，会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMillis 以内。如果读者希望减少停顿时间，而把这个值设得很小，为了达到预期的停顿时间，虚拟机可能会使用一个较小的堆（一个小堆比一个大堆回收快），而这将导致垃圾回收变得很频繁，从而增加了垃圾回收总时间，降低了吞吐量。
- –XX:GCTimeRatio：设置吞吐量大小。它的值是一个 0 到 100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 $1/(1+n)$的时间用于垃圾收集。比如 GCTimeRatio 等于 19（默认值），则系统用于垃圾收集的时间不超过 1/(1+19)=5%。默认情况下，它的取值是 99，即不超过 1/(1+99)=1%的时间用于垃圾收集。

除此以外，ParallelGC 回收器与 ParNew 回收器另一个不同之处在于它还支持一种自适应的 GC 调节策略。使用-XX:+UseAdaptiveSizePolicy 可以打开自适应 GC 策略。在这种模式下，新生代的大小、eden 和 survivior 的比例、晋升老年代的对象年龄等参数会被自动调整，以达到在堆大小、吞吐量和停顿时间之间的平衡点。在手工调优比较困难的场合，可以直接使用这种自适应的方式，仅指定虚拟机的最大堆、目标吞吐量（GCTimeRatio）和停顿时间（MaxGCPauseMillis），让虚拟机自己完成调优工作。

ParallelGC 回收器的工作日志如下所示：

```
  0.880: [GC [PSYoungGen: 16448K->2439K(19136K)] 16448K->2439K(62848K), 0.0064912
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

它显示了 ParallelGC 回收器的工作成果，也就是回收前的内存大小和回收后的内存大小，以及花费的时间。

注意：ParallelGC 回收器关注系统吞吐量。可以通过-XX:MaxGCPauseMillis 和-XX:GCTimeRatio 设置期望的停顿时间和吞吐量大小。但是鱼和熊掌不可兼得，这两个参数是相互矛盾的，通常如果减少一次收集的最大停顿时间，就会同时减少系统吞吐量，增加系统吞吐量又可能会同时增加一次垃圾回收的最大停顿。

5.2.3　老年代 ParallelOldGC 回收器

老年代 ParallelOldGC 回收器也是一种多线程并发的收集器。和新生代 ParallelGC 回收器一样，它也是一种关注吞吐量的收集器。从名字上看，它在 ParallelGC 中间插入了 Old，表示这是一个应用于老年代的回收器，并且和 ParallelGC 新生代回收器搭配使用。

ParallelOldGC 回收器使用标记压缩算法，它在 JDK1.6 中才可以使用。图 5.3 显示了老年代 ParallelOldGC 回收器的工作模式。

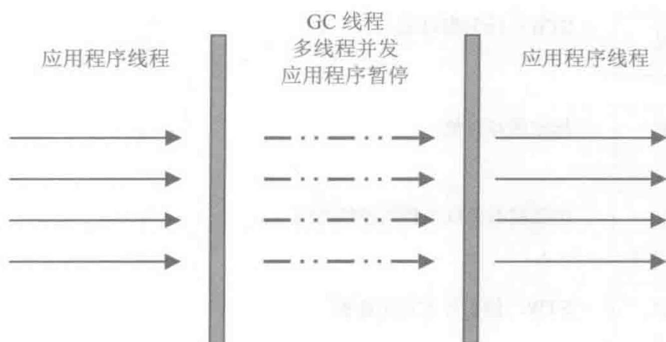


图 5.3　老年代 ParallelOldGC 回收器工作示意图

使用-XX:+UseParallelOldGC 可以在新生代使用 ParallelGC 回收器，老年代使用 ParallelOldGC 回收器。这是一对非常关注吞吐量的垃圾回收器组合。在对吞吐量敏感的系统中，可以考虑使用。参数-XX:ParallelGCThreads 也可以用于设置垃圾回收时的线程数量。

ParallelOldGC 回收器的工作日志如下：

```
1.500: [Full GC [PSYoungGen: 2682K->0K(19136K)] [ParOldGen: 28035K->30437K(43712K)] 30717K->30437K(62848K) [PSPermGen: 10943K->10928K(32768K)], 0.2902791 secs] [Times: user=1.44 sys=0.03, real=0.30 secs]
```

它显示了新生代、老年代以及永久区在回收前后的情况，以及 Full GC 所消耗的时间。

5.3 一心多用都不落下：CMS 回收器

与 ParallelGC 和 ParallelOldGC 不同，CMS 回收器主要关注于系统停顿时间。CMS 是 Concurrent Mark Sweep 的缩写，意为并发标记清除，从名称上就可以得知，它使用的是标记清除算法，同时它又是一个使用多线程并行回收的垃圾回收器。

5.3.1 CMS 主要工作步骤

CMS 回收器的工作过程与其他垃圾收集器相比，略显复杂。CMS 工作时，主要步骤有：初始标记、并发标记、预清理、重新标记、并发清除和并发重置。其中初始标记和重新标记是独占系统资源的，而预清理、并发标记、并发清除和并发重置是可以和用户线程一起执行的。因此，从整体上说，CSM 收集不是独占式的，它可以在应用程序运行过程中进行垃圾回收。CMS 回收器的工作流程如图 5.4 所示。

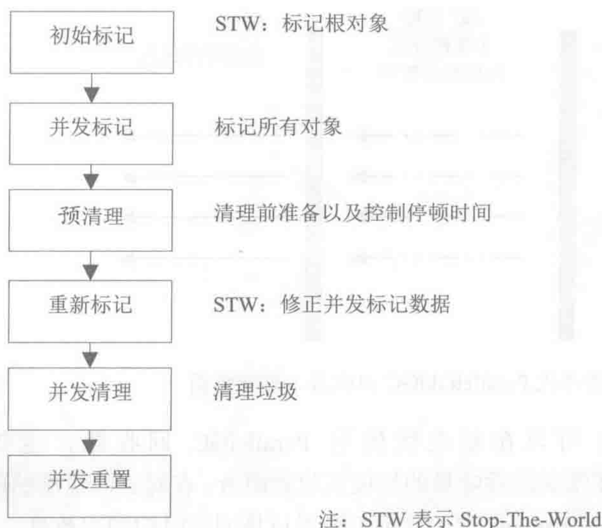


图 5.4 CMS 工作流程示意图

根据标记清除算法，初始标记、并发标记和重新标记都是为了标记出需要回收的对象。并发清理则是在标记完成后，正式回收垃圾对象。并发重置是指在垃圾回收完成后，重新初始化 CMS 数据结构和数据，为下一次垃圾回收做好准备。并发标记、并发清理和并发重置都是可以和应用程序线程一起执行的。

在整个 CMS 回收过程中，默认情况下，在并发标记之后，会有一个预清理的操作（也可以关闭开关-XX:-CMSPrecleaningEnabled，不进行预清理）。预清理是并发的，除了为正式清理做准备和检查以外，预清理还会尝试控制一次停顿时间。由于重新标记是独占 CPU 的，如果新生代 GC 发生后，立即触发一次重新标记，那么一次停顿时间可能很长。为了避免这种情况，预处理时，会刻意等待一次新生代 GC 的发生，然后根据历史性能数据预测下一次新生代 GC 可能发生的时间，然后在当前时间和预测时间的中间时刻，进行重新标记。这样，从最大程度上避免新生代 GC 和重新标记重合，尽可能减少一次停顿时间。

5.3.2　CMS 主要的设置参数

启用 CMS 回收器的参数是-XX:+UseConcMarkSweepGC。CMS 是多线程回收器，设置合理的工作线程数量也对系统性能有重要的影响。

CMS 默认启动的并发线程数是(ParallelGCThreads+3)/4)。ParallelGCThreads 表示 GC 并行时使用的线程数量，如果新生代使用 ParNew，那么 ParallelGCThreads 也就是新生代 GC 的线程数量。这意味着有 4 个 ParallelGCThreads 时，只有 1 个并发线程，而两个并发线程时，有 5~8 个 ParallelGCThreads 线程数。

并发线程数量也可以通过-XX:ConcGCThreads 或者-XX:ParallelCMSThreads 参数手工设定。当 CPU 资源比较紧张时，受到 CMS 回收器线程的影响，应用系统的性能在垃圾回收阶段可能会非常糟糕。

> 注意：并发是指收集器和应用线程交替执行，并行是指应用程序停止，同时由多个线程一起执行 GC。因此并行回收器不是并发的。因为并行回收器执行时，应用程序完全挂起，不存在交替执行的步骤。

由于 CMS 回收器不是独占式的回收器，在 CMS 回收过程中，应用程序仍然在不停地工作。在应用程序工作过程中，又会不断地产生垃圾。这些新生成的垃圾在当前 CMS 回收过程中是无法清除的。同时，因为应用程序没有中断，所以在 CMS 回收过程中，还应该确保应用程序有足够的内存可用。因此，CMS 回收器不会等待堆内存饱和时才进行垃圾回收，而是当堆内存使用率达到某一阈值时便开始进行回收，以确保应用程序在 CMS 工作过程中，依然有足够的空间支持应用程序运行。

这个回收阈值可以使用-XX:CMSInitiatingOccupancyFraction 来指定，默认是 68。即当老年代的空间使用率达到 68%时，会执行一次 CMS 回收。如果应用程序的内存使用率增长很快，在 CMS 的执行过程中，已经出现了内存不足的情况，此时，CMS 回收就会失败，虚拟机将启

动老年代串行收集器进行垃圾回收。如果这样，应用程序将完全中断，直到垃圾回收完成，这时，应用程序的停顿时间可能会较长。

> 注意：通过-XX:CMSInitiatingOccupancyFraction 可以指定当老年代空间使用率达到多少时，进行一次 CMS 垃圾回收。

因此，根据应用程序的特点，可以对-XX:CMSInitiatingOccupancyFraction 进行调优。如果内存增长缓慢，则可以设置一个稍大的值，大的阈值可以有效降低 CMS 的触发频率，减少老年代回收的次数可以较为明显地改善应用程序性能。反之，如果应用程序内存使用率增长很快，则应该降低这个阈值，以避免频繁触发老年代串行收集器。

CMS 是一个基于标记清除算法的回收器。在本章之前的篇幅中已经提到，标记清除算法将会造成大量内存碎片，离散的可用空间无法分配较大的对象。图 5.5 显示了 CMS 回收前后老年代的情况。

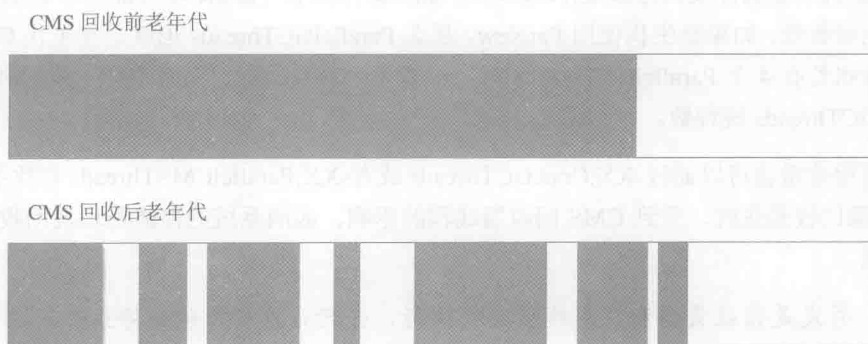


图 5.5　CMS 回收前后老年代对比示意图

在这种情况下，即使堆内存仍然有较大的剩余空间，也可能会被迫进行一次垃圾回收，以换取一块可用的连续内存。这种现象对系统性能是相当不利的，为了解决这个问题，CMS 回收器还提供了几个用于内存压缩整理的参数。

-XX:+UseCMSCompactAtFullCollection 开关可以使 CMS 在垃圾收集完成后，进行一次内存碎片整理，内存碎片的整理不是并发进行的。-XX:CMSFullGCsBeforeCompaction 参数可以用于设定进行多少次 CMS 回收后，进行一次内存压缩。

5.3.3 CMS 的日志分析

CMS 回收器工作时的日志输出如下所示：

```
1.313: [GC [1 CMS-initial-mark: 69112K(136576K)] 77037K(198016K), 0.0120453 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
1.325: [CMS-concurrent-mark-start]
1.342: [GC1.342: [ParNew: 61440K->4557K(61440K), 0.0025283 secs] 130552K->74933K(198016K), 0.0026144 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.365: [GC1.365: [ParNew: 59213K->5972K(61440K), 0.0036258 secs] 129589K->77612K(198016K), 0.0037053 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.388: [GC1.388: [ParNew: 60628K->6468K(61440K), 0.0025254 secs] 132268K->79371K(198016K), 0.0026154 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.406: [CMS-concurrent-mark: 0.072/0.082 secs] [Times: user=0.17 sys=0.00, real=0.08 secs]
1.406: [CMS-concurrent-preclean-start]
1.409: [CMS-concurrent-preclean: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.409: [CMS-concurrent-abortable-preclean-start]
1.411: [GC1.411: [ParNew: 61124K->4691K(61440K), 0.0025330 secs] 134027K->78858K(198016K), 0.0026144 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.422: [CMS-concurrent-abortable-preclean: 0.011/0.014 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
1.423: [GC[YG occupancy: 35483 K (61440 K)]1.423: [Rescan (parallel) , 0.0102064 secs]1.433: [weak refs processing, 0.0000142 secs]1.433: [scrub string table, 0.0000298 secs] [1 CMS-remark: 74166K(136576K)] 109650K(198016K), 0.0103386 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
1.433: [CMS-concurrent-sweep-start]
1.445: [GC1.445: [ParNew: 59347K->6001K(61440K), 0.0037510 secs] 132617K->80535K(198016K), 0.0038390 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.450: [CMS-concurrent-sweep: 0.013/0.017 secs] [Times: user=0.06 sys=0.00, real=0.02 secs]
1.450: [CMS-concurrent-reset-start]
1.451: [CMS-concurrent-reset: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

以上信息是一次 CMS 收集的输出。可以看到，在 CMS 回收器的工作过程中，包括了初始化标记、并发标记、预清理、重新标记、并发清理和重发重置等几个重要阶段。在日志中，还可以看到 CMS 的耗时以及堆内存信息。

在 1.409 秒时，发生 abortable-preclean，表示 CMS 开始等待一次新生代 GC。在 1.411 秒，ParNew 回收器工作。在 1.422 秒，abortable-preclean 终止。之后，CMS 根据之前新生代 GC 的情况，将重新标记的时间放置在一个最不可能和下一次新生代 GC 重叠的时刻，通常为两次新生代 GC 的中间点，这里为 1.423 秒处。

除此之外，CMS 回收器在运行时，还可能会输出如下日志：

```
    33.348: [Full GC 33.348: [CMS33.357: [CMS-concurrent-sweep: 0.035/0.036 secs]
[Times: user=0.11 sys=0.03, real=0.03 secs]
    (concurrent mode failure): 47066K->39901K(49152K), 0.3896802 secs] 60771K->
39901K(63936K), [CMS Perm : 22529K->22529K(32768K)], 0.3897989 secs] [Times:
user=0.39 sys=0.00, real=0.39 secs]
```

这说明 CMS 回收器并发收集失败。这很可能是由于应用程序在运行过程中老年代空间不够所导致。如果在 CMS 工作过程中，出现非常频繁的并发模式失败，就应该考虑进行调整，尽可能预留一个较大的老年代空间。或者可以设置一个较小的-XX:CMSInitiatingOccupancyFraction 参数，降低 CMS 触发的阈值，使 CMS 在执行过程中，仍然有较大的老年代空闲空间供应用程序使用。

注意： CMS 回收器是一个关注停顿的垃圾收集器。同时 CMS 回收器在部分工作流程中，可以与用户程序同时运行，从而降低应用程序的停顿时间。

5.3.4 有关 Class 的回收

在使用 CMS 回收器时，如果需要回收 Perm 区，那么默认情况下，还是需要触发一次 Full GC 的，如下所示：

```
    [Full GC[CMS: 4624K->2047K(10944K), 0.0156648 secs] 7191K->2047K(15936K),
[CMS Perm : 4096K->2145K(4096K)], 0.0157072 secs] [Times: user=0.01 sys=0.00,
real=0.02 secs]
```

如果希望使用 CMS 回收 Perm 区，则必须要打开-XX:+ CMSClassUnloadingEnabled 开关。使用-XX:+ CMSClassUnloadingEnabled 后，如果条件允许，那么系统会使用 CMS 的机制回收 Perm 区 Class 数据，如下日志：

```
    [GC [1 CMS-initial-mark: 6035K(10944K)] 6640K(15936K), 0.0007878 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
    [CMS-concurrent-mark: 0.008/0.009 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
    [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
    [GC[YG occupancy: 1070 K (4992 K)][Rescan (parallel) , 0.0008064 secs][weak refs
processing, 0.0001239 secs][class unloading, 0.0016846 secs][scrub symbol table,
0.0004835 secs][scrub string table, 0.0000285 secs] [1 CMS-remark: 6035K(10944K)]
7105K(15936K), 0.0033860 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
    [CMS-concurrent-sweep: 0.009/0.009 secs] [Times: user=0.00 sys=0.00, real=0.01
secs]
```

```
[CMS-concurrent-reset: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
```

5.4 未来我做主：G1 回收器

G1 回收器（Garbage-First）是在 JDK 1.7 中正式使用的全新的垃圾回收器，从长期目标来看，它是为了取代 CMS 回收器。G1 回收器拥有独特的垃圾回收策略，这和之前提到的回收器截然不同。从分代上看，G1 依然属于分代垃圾回收器，它会区分年轻代和老年代，依然有 eden 区和 survivor 区，但从堆的结构上看，它并不要求整个 eden 区、年轻代或者老年代都连续。它使用了分区算法（有关分区算法的简介，参考第 4 章）。作为 CMS 的长期替代方案，G1 同时使用了全新的分区算法，其特点如下。

- **并行性**：G1 在回收期间，可以由多个 GC 线程同时工作，有效利用多核计算能力。
- **并发性**：G1 拥有与应用程序交替执行的能力，部分工作可以和应用程序同时执行，因此一般来说，不会在整个回收期间完全阻塞应用程序。
- **分代 GC**：G1 依然是一个分代收集器，但是和之前回收器不同，它同时兼顾年轻代和老年代。对比其他回收器，它们或者工作在年轻代，或者工作在老年代。因此，这里是一个很大的不同。
- **空间整理**：G1 在回收过程中，会进行适当的对象移动，不像 CMS，只是简单地标记清理对象，在若干次 GC 后，CMS 必须进行一次碎片整理。而 G1 不同，它每次回收都会有效地复制对象，减少空间碎片。
- **可预见性**：由于分区的原因，G1 可以只选取部分区域进行内存回收，这样缩小了回收的范围，因此对于全局停顿也能得到较好的控制。

5.4.1 G1 的内存划分和主要收集过程

G1 收集器将堆进行分区，划分为一个个的区域，每次收集的时候，只收集其中几个区域，以此来控制垃圾回收产生的一次停顿时间。

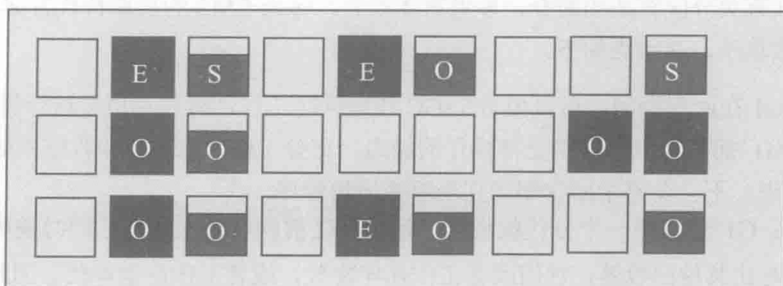
G1 的收集过程可能有 4 个阶段：

- 新生代 GC
- 并发标记周期
- 混合收集
- 如果需要，可能会进行 Full GC

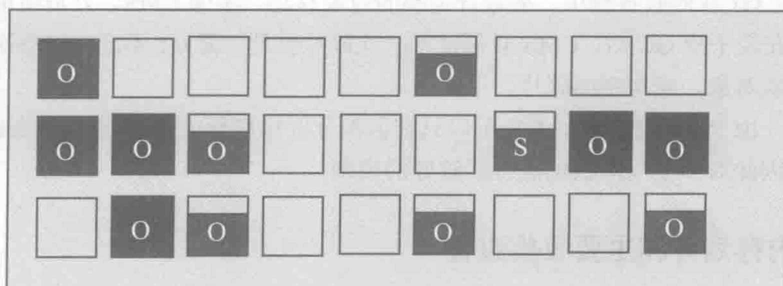
5.4.2 G1 的新生代 GC

新生代 GC 的主要工作是回收 eden 区和 survivor 区。一旦 eden 区被占满，新生代 GC 就会启动。新生代 GC 收集前后的堆数据如图 5.6 所示，其中 E 表示 eden 区，S 表示 survivor 区，O 表示老年代。可以看到，新生代 GC 只处理 eden 和 survivor 区，回收后，所有的 eden 区都应该被清空，而 survivor 区会被收集一部分数据，但是应该至少仍然存在一个 survivor 区，类比其他的新生代收集器，这一点似乎并没有太大变化。另一个重要的变化是老年代的区域增多，因为部分 survivor 区或者 eden 区的对象可能会晋升到老年代。

新生代 GC 前



新生代 GC 后



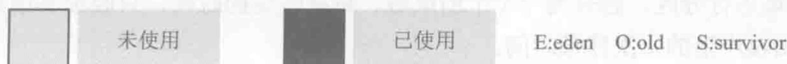


图 5.6 G1 的新生代 GC

新生代 GC 发生后，如果打开了 PrintGCDetails 选项，就可以得到类似如下的 GC 日志（这里只给出了部分日志，完全的日志及其分析请看 5.4.6 节）：

```
0.336: [GC pause (young), 0.0063051 secs]
....
```

```
    [Eden: 235.0M(235.0M)->0.0B(229.0M) Survivors: 5120.0K->11.0M Heap:
239.2M(400.0M)->10.5M(400.0M)]
   [Times: user=0.06 sys=0.00, real=0.01 secs]
```

和其他回收器的日志相比，G1 的日志内容非常丰富。当然我们最为关心的依然是 GC 的停顿时间以及回收情况。从日志中可以看到，eden 区原本占用 235MB 空间，回收后被清空，survivor 区从 5MB 增长到了 11MB，这是因为部分对象从 eden 区复制到 survivor 区，整个堆合计为 400MB，从回收前的 239MB 下降到 10.5MB。

5.4.3　G1 的并发标记周期

G1 的并发阶段和 CMS 有点类似，它们都是为了降低一次停顿时间，而将可以和应用程序并发的部分单独提取出来执行。

并发标记周期可以分为以下几步。

- **初始标记**：标记从根节点直接可达的对象。这个阶段会伴随一次新生代 GC，它是会产生全局停顿的，应用程序线程在这个阶段必须停止执行。
- **根区域扫描**：由于初始标记必然会伴随一次新生代 GC，所以在初始化标记后，eden 被清空，并且存活对象被移入 survivor 区。在这个阶段，将扫描由 survivor 区直接可达的老年代区域，并标记这些直接可达的对象。这个过程是可以和应用程序并发执行的。但是根区域扫描不能和新生代 GC 同时执行（因为根区域扫描依赖 survivor 区的对象，而新生代 GC 会修改这个区域），因此如果恰巧在此时需要进行新生代 GC，GC 就需要等待根区域扫描结束后才能进行，如果发生这种情况，这次新生代 GC 的时间就会延长。
- **并发标记**：和 CMS 类似，并发标记将会扫描并查找整个堆的存活对象，并做好标记。这是一个并发的过程，并且这个过程可以被一次新生代 GC 打断。
- **重新标记**：和 CMS 一样，重新标记也是会产生应用程序停顿的。由于在并发标记过程中，应用程序依然在运行，因此标记结果可能需要进行修正，所以在此对上一次的标记结果进行补充。在 G1 中，这个过程使用 SATB（Snapshot-At-The-Beginning）算法完成，即 G1 会在标记之初为存活对象创建一个快照，这个快照有助于加速重新标记的速度。
- **独占清理**：这个阶段是会引起停顿的。它将计算各个区域的存活对象和 GC 回收比例并进行排序，识别可供混合回收的区域。在这个阶段，还会更新记忆集（Remebered Set）。该阶段给出了需要被混合回收的区域并进行了标记，在混合回收阶段，需要这些信息。
- **并发清理阶段**：这里会识别并清理完全空闲的区域。它是并发的清理，不会引起停顿。

图 5.7 显示了并发标记周期前后堆的可能的情况。由于并发标记周期包含一次新生代 GC，故新生会被整理，但由于并发标记周期执行时，应用程序依然在运行，因此，并发标记周期结束后，又会有新的 eden 空间被使用。并发标记周期执行前后最大的不同是在该阶段后，系统增加了一些标记为 G 的区域。这些区域被标记，是因为它们内部的垃圾比例较高，因此希望在后续的混合 GC 中进行收集（注意在并发标记周期中并未正式收集这些区域）。这些将要被回收的区域会被 G1 记录在一个称为 Collection Sets（回收集）的集合中。

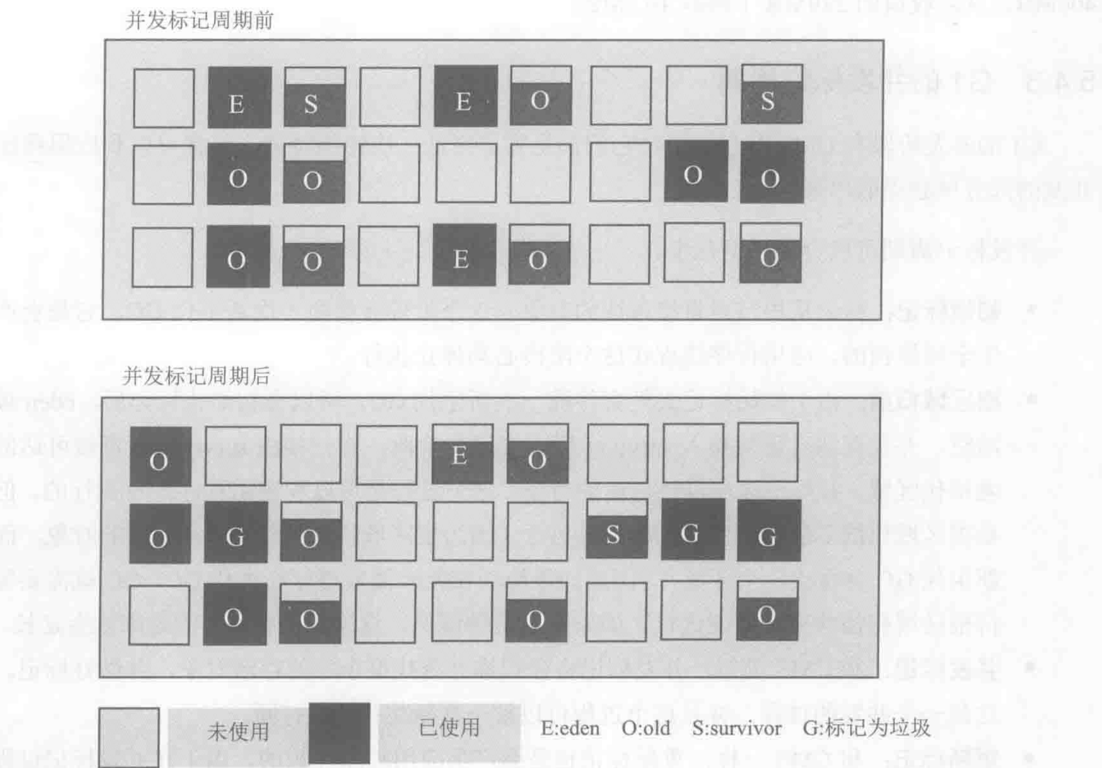


图 5.7 并发回收阶段前后的可能情况

并发回收阶段的整体工作流程如图 5.8 所示，可以看到除了初始标记、重新标记和独占清理外，其他几个阶段都可以和应用程序并发执行。

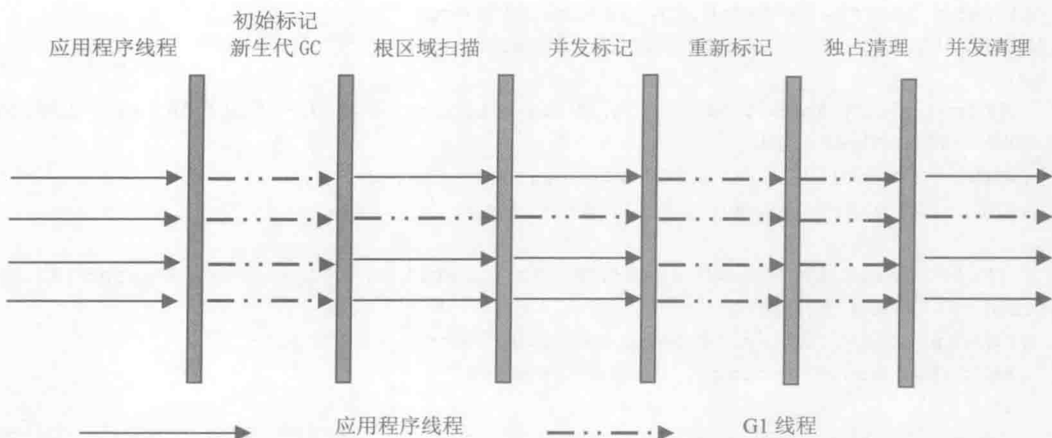


图 5.8　G1 并发标记周期的主要工作流程

在并发标记周期中，G1 会产生如下日志。

（1）初始标记，它伴随一次新生代 GC。

```
1.765: [GC pause (young) (initial-mark), 0.0052941 secs]
…
   [Eden: 61.0M(61.0M)->0.0B(57.0M) Survivors: 3072.0K->5120.0K Heap: 174.8M
(200.0M)->116.3M(200.0M)]
  [Times: user=0.00 sys=0.00, real=0.01 secs]
```

可以看到，初始化标记时，eden 区被清空，并部分复制到 survivor 区。

（2）是一次并发的根区域扫描，并发扫描过程不能被新生代 GC 中断。

```
1.771: [GC concurrent-root-region-scan-start]
1.772: [GC concurrent-root-region-scan-end, 0.0011676 secs]
```

根区域扫描不会产生停顿。

（3）就是并发标记，并发标记可以被新生代 GC 打断，下面的日志显示了一次并发标记被 3 次新生代 GC 打断。

```
1.772: [GC concurrent-mark-start]
1.793: [GC pause (young), 0.0057816 secs]
...
   [Eden: 57.0M(57.0M)->0.0B(55.0M) Survivors: 5120.0K->6144.0K Heap: 173.3M
(200.0M)->117.5M(200.0M)]
```

```
 [Times: user=0.00 sys=0.00, real=0.01 secs]
1.818: [GC pause (young), 0.0062386 secs]
...
   [Eden: 55.0M(55.0M)->0.0B(55.0M) Survivors: 6144.0K->5120.0K Heap: 172.5M
(200.0M)->118.2M(200.0M)]
 [Times: user=0.00 sys=0.00, real=0.01 secs]
1.842: [GC pause (young), 0.0068966 secs]
...
   [Eden: 55.0M(55.0M)->0.0B(53.0M) Survivors: 5120.0K->6144.0K Heap: 173.2M
(200.0M)->119.2M(200.0M)]
 [Times: user=0.00 sys=0.00, real=0.01 secs]
1.867: [GC pause (young), 0.0067426 secs]
...
   [Eden: 53.0M(53.0M)->0.0B(52.0M) Survivors: 6144.0K->6144.0K Heap: 172.2M
(200.0M)->120.7M(200.0M)]
 [Times: user=0.06 sys=0.00, real=0.01 secs]
1.878: [GC concurrent-mark-end, 0.1055874 secs]
```

（4）重新标记是会引起全局停顿的，它的日志如下：

```
1.878: [GC remark 1.878: [GC ref-proc, 0.0000447 secs], 0.0015223 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
```

（5）重新标记后会进行独占清理，独占清理会重新计算各个区域的存活对象，并以此可以得到每个区域进行 GC 的效用（即回收比）。它的日志如下：

```
1.879: [GC cleanup 127M->127M(200M), 0.0009723 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
```

（6）并发清理是并发执行的，它会根据独占清理阶段计算得出的每个区域的存活对象数量，直接回收已经不包含存活对象的区域。它的日志如下：

```
1.498: [GC concurrent-cleanup-start]
1.498: [GC concurrent-cleanup-end, 0.0000073 secs]
```

5.4.4 混合回收

在并发标记周期中，虽然有部分对象被回收，但是总体上说，回收的比例是相当低的。但是在并发标记周期后，G1 已经明确知道哪些区域含有比较多的垃圾对象，在混合回收阶段，就可以专门针对这些区域进行回收。当然，G1 会优先回收垃圾比例较高的区域，因为回收这些区域的性价比也比较高。而这也正是 G1 名字的由来。G1 全称为 Garbage First Garbage Collector，直译为垃圾优先的垃圾回收器，这里的垃圾优先（Garbage First）指的就是回收时优先选取垃圾

比例最高的区域。

这个阶段叫作混合回收，是因为在这个阶段，既会执行正常的年轻代 GC，又会选取一些被标记的老年代区域进行回收，它同时处理了新生代和老年代，如图 5.9 所示。因为新生代 GC 的原因，eden 区域必然被清空，此外，有两块被标记位 G 的垃圾比例最高的区域被清理。被清理区域中的存活对象会被移动到其他区域，这样的好处是可以减少空间碎片。

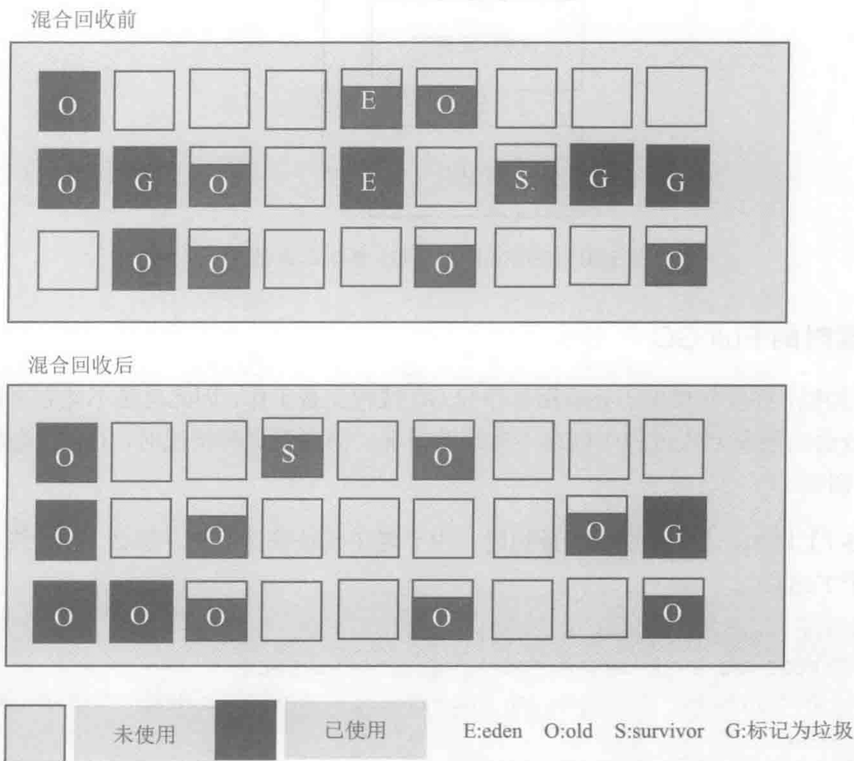


图 5.9　G1 的混合回收

混合 GC 会产生如下日志：

```
  1.904: [GC pause (mixed), 0.0073135 secs]
  ...
     [Eden: 4096.0K(4096.0K)->0.0B(53.0M) Survivors: 6144.0K->2048.0K Heap:
127.3M(200.0M)->123.6M(200.0M)]
   [Times: user=0.00 sys=0.00, real=0.01 secs]
```

混合 GC 会执行多次，直到回收了足够多的内存空间，然后，它会触发一次新生代 GC。新

生代 GC 后，又可能会发生一次并发标记周期的处理，最后，又会引起混合 GC 的执行。因此整个过程可能如图 5.10 所示。



图 5.10　混合 GC 以及 G1 整体示意图

5.4.5　必要时的 Full GC

和 CMS 类似，并发收集由于让应用程序和 GC 线程交替工作，因此总是不能完全避免在特别繁忙的场合会出现在回收过程中内存不充足的情况。当遇到这种情况时，G1 也会转入一个 Full GC 进行回收。

【示例 5-1】比如，当 G1 在并发标记时，由于老年代被快速填充，那么 G1 会终止并发标记而转入一个 Full GC。

```
   24.909: [GC concurrent-mark-start]
   24.909: [Full GC 898M->896M(900M), 0.7505595 secs]
     [Eden: 0.0B(45.0M)->0.0B(45.0M) Survivors: 0.0B->0.0B Heap: 898.7M(900.0M)->
896.2M(900.0M)]
    [Times: user=1.05 sys=0.00, real=0.75 secs]
   25.660: [GC concurrent-mark-abort]
```

此外，如果在混合 GC 时发生空间不足或者在新生代 GC 时，survivor 区和老年代无法容纳幸存对象，都会导致一次 Full GC 产生。

5.4.6　G1 日志

G1 的日志与先前的回收器相比已经丰富了很多。在本书前文中，尚未给出完整的 G1 日志。本节将给出一个较为完整的日志，并做出解释。

【示例 5-2】以下是一个完整的 G1 新生代日志。

```
 1.619: [GC pause (young) (initial-mark), 0.03848843 secs]
    [Parallel Time:  38.0 ms]
       [GC Worker Start (ms):  1619.3  1619.3  1619.3  1619.3
        Avg: 1619.3, Min: 1619.3, Max: 1619.3, Diff:   0.0]
       [Ext Root Scanning (ms):  0.3  0.3  0.2  0.2
        Avg:   0.3, Min:   0.2, Max:   0.3, Diff:   0.1]
       [Update RS (ms):  5.7  5.4  28.0  5.3
        Avg:  11.1, Min:   5.3, Max:  28.0, Diff:  22.8]
          [Processed Buffers : 5 4 1 4
           Sum: 14, Avg: 3, Min: 1, Max: 5, Diff: 4]
       [Scan RS (ms):  4.6  5.0  0.0  5.2
        Avg:   3.7, Min:   0.0, Max:   5.2, Diff:   5.2]
       [Object Copy (ms):  27.4  27.3  9.6  27.2
        Avg:  22.9, Min:   9.6, Max:  27.4, Diff:  17.7]
       [Termination (ms):  0.1  0.0  0.0  0.1
        Avg:   0.0, Min:   0.0, Max:   0.1, Diff:   0.1]
          [Termination Attempts : 3 1 10 5
           Sum: 19, Avg: 4, Min: 1, Max: 10, Diff: 9]
       [GC Worker End (ms):  1657.3  1657.2  1657.2  1657.2
        Avg: 1657.2, Min: 1657.2, Max: 1657.3, Diff:   0.0]
       [GC Worker (ms):  38.0  38.0  38.0  38.0
        Avg:  38.0, Min:  38.0, Max:  38.0, Diff:   0.1]
       [GC Worker Other (ms):  0.0  0.1  0.1  0.1
        Avg:   0.1, Min:   0.0, Max:   0.1, Diff:   0.1]
    [Clear CT:   0.0 ms]
    [Other:  0.4 ms]
       [Choose CSet:   0.0 ms]
       [Ref Proc:   0.1 ms]
       [Ref Enq:   0.0 ms]
       [Free CSet:   0.1 ms]
    [Eden: 32M(35M)->0B(35M) Survivors: 5120K->5120K Heap: 147M(200M)->147M
(200M)]
  [Times: user=0.16 sys=0.00, real=0.04 secs]
```

（1）日志第一行：

```
 1.619: [GC pause (young) (initial-mark), 0.03848843 secs]
```

表示在应用程序开启 1.619 秒时发生了一次新生代 GC，这是在初始标记时发生的，耗时 0.038 秒，意味着应用程序至少暂停了 0.038 秒。

（2）后续并行时间：

```
[Parallel Time:  38.0 ms]
```

表示所有 GC 线程总的花费时间，这里为 38 毫秒。

（3）给出每一个 GC 线程的执行情况：

```
[GC Worker Start (ms):  1619.3  1619.3  1619.3  1619.3
  Avg: 1619.3, Min: 1619.3, Max: 1619.3, Diff:   0.0]
```

这里表示一共 4 个 GC 线程（因为第一行有 4 个数据），它们都在 1619.3 秒时启动。同时，还给出了这几个启动数据的统计值，如平均（Avg），最小（Min）、最大（Max）和差值（Diff）。Diff 表示最大值和最小值的差。

（4）给出了根扫描的耗时：

```
[Ext Root Scanning (ms):  0.3  0.3  0.2  0.2
 Avg:   0.3, Min:   0.2, Max:   0.3, Diff:   0.1]
```

如上，在根扫描时（全局变量、系统数据字典、线程栈等），每一个 GC 线程的耗时，这里分配消耗了 0.3、0.3、0.2、0.2 秒时间，后一行给出这些耗时的统计数据。

（5）给出了更新记忆集（Remembered Sets）的耗时：

```
[Update RS (ms):  5.7  5.4  28.0  5.3
  Avg:  11.1, Min:   5.3, Max:  28.0, Diff:  22.8]
    [Processed Buffers : 5 4 1 4
      Sum: 14, Avg: 3, Min: 1, Max: 5, Diff: 4]
```

记忆集是 G1 中维护的一个数据结构，简称 RS。每一个 G1 区域都有一个 RS 与之关联。由于 G1 回收时，是按照区域回收的，比如在回收区域 A 的对象时，很可能并不回收区域 B 的对象，此时，为了回收区域 A 的对象，要扫描区域 B 甚至是整个堆来判定区域 A 中哪些对象不可达，这样做的代价显然很大。因此，G1 在区域 A 的 RS 中，记录了在区域 A 中被其他区域引用的对象，这样在回收区域 A 时，只要将 RS 视为区域 A 根集的一部分即可，从而可以避免做整个堆的扫描。由于系统在运行过程中，对象之间的引用关系是可能时刻变化的，因此为了更高效地跟踪这些引用关系，会将这些变化记录在 Update Buffers 中。这里的 Processed Buffers 指的就是处理这个 Update Buffers 数据。这里给出的 4 个时间和也是 4 个 GC 线程的耗时，以及它们的统计数据。从这个日志中可以看到，更新 RS 时，分别耗时 5.7、5.4、28、5.3 毫秒，平均耗时 11.1 毫秒。

（6）扫描 RS 的时间：

```
[Scan RS (ms):  4.6  5.0  0.0  5.2
```

```
      Avg:   3.7, Min:   0.0, Max:   5.2, Diff:   5.2]
```

（7）在正式回收时，G1 会对被回收区域的对象进行疏散，即将存活对象放置在其他区域中，因此需要进行对象的复制。

```
 [Object Copy (ms):  27.4  27.3  9.6  27.2
   Avg:  22.9, Min:   9.6, Max:  27.4, Diff:  17.7]
```

这里给出的 Object Copy 就是进行对象赋值的耗时。

（8）给出了 GC 工作线程的终止信息：

```
 [Termination (ms):  0.1  0.0  0.0  0.1
    Avg:   0.0, Min:   0.0, Max:   0.1, Diff:   0.1]
      [Termination Attempts : 3 1 10 5
         Sum: 19, Avg: 4, Min: 1, Max: 10, Diff: 9]
```

这里的终止时间是线程花在终止阶段的耗时。在 GC 线程终止前，它们会检查其他 GC 线程的工作队列，查看是否仍然还有对象引用没有处理完，如果其他线程仍然有没有处理完的数据，请求终止的 GC 线程就会帮助它尽快完成，随后，再尝试终止。其中 Termination Attempts 展示了每一个工作线程尝试终止的次数。

（9）显示了 GC 工作线程的完成时间：

```
 [GC Worker End (ms):  1657.3  1657.2  1657.2  1657.2
      Avg: 1657.2, Min: 1657.2, Max: 1657.3, Diff:   0.0]
```

这里显示了在系统运行后 1657 毫秒附近，这几个线程都终止了。

（10）显示了几个 GC 工作线程的存活时间，单位是毫秒：

```
 [GC Worker (ms):  38.0  38.0  38.0  38.0
      Avg:  38.0, Min:  38.0, Max:  38.0, Diff:   0.1]
```

（11）是 GC 线程花费在其他任务中的耗时，单位是毫秒，可以看到这部分时间非常少：

```
 [GC Worker Other (ms):  0.0  0.1  0.1  0.1
      Avg:   0.1, Min:   0.0, Max:   0.1, Diff:   0.1]
```

（12）是清空 CardTable 的时间，RS 就是依靠 CardTable 来记录哪些是存活对象的：

```
 [Clear CT:   0.0 ms]
```

（13）显示了其他几个任务的耗时：

```
 [Other:   0.4 ms]
   [Choose CSet:   0.0 ms]
```

```
      [Ref Proc:   0.1 ms]
      [Ref Enq:   0.0 ms]
   [Free CSet:   0.1 ms]
```

比如选择 CSet（Collection Sets）的时间、Ref Proc（处理弱引用、软引用的时间）、Ref Enq（弱引用、软引用入队时间）和 Free CSet（释放被回收的 CSet 中区域的时间，包括它们的 RS）。

注意：Collection Sets 表示被选取的、将要被收集的区域的集合。

（14）最后，就是比较熟悉的 GC 回收的整体情况：

```
    [Eden: 32M(35M)->0B(35M) Survivors: 5120K->5120K Heap: 147M(200M)->147M(200M)]
  [Times: user=0.16 sys=0.00, real=0.04 secs]
```

这里显示了 eden 区一共 32MB 被清空，survivor 区没有释放对象，整个堆空间没有释放空间。用户 CPU 耗时 0.16 秒，实际耗时 0.04 秒。

5.4.7 G1 相关的参数

对于 G1 收集器，可以使用-XX:+UseG1GC 标记打开 G1 收集器开关，对 G1 收集器进行设置时，最重要的一个参数就是-XX:MaxGCPauseMillis，它用于指定目标最大停顿时间。如果任何一次停顿超过这个设置值时，G1 就会尝试调整新生代和老年代的比例、调整堆大小、调整晋升年龄等手段，试图达到预设目标。对于性能调优来说，有时候，总是鱼和熊掌不可兼得的，如果停顿时间缩短，对于新生代来说，这意味着很可能要增加新生代 GC 的次数，GC 反而会变得更加频繁。对于老年代区域来说，为了获得更短的停顿时间，那么在混合 GC 收集时，一次收集的区域数量也会变少，这样无疑增加了进行 Full GC 的可能性。

另外一个重要的参数是-XX:ParallelGCThreads，它用于设置并行回收时，GC 的工作线程数量。

此外，-XX:InitiatingHeapOccupancyPercent 参数可以指定当整个堆使用率达到多少时，触发并发标记周期的执行。默认值是 45，即当整个堆占用率达到 45%时，执行并发标记周期。InitiatingHeapOccupancyPercent 一旦设置，始终都不会被 G1 收集器修改，这意味着 G1 收集器不会试图改变这个值，来满足 MaxGCPauseMillis 的目标。如果 InitiatingHeapOccupancyPercent 值设置偏大，会导致并发周期迟迟得不到启动，那么引起 Full GC 的可能性也大大增加，反之，一个过小的 InitiatingHeapOccupancyPercent 值，会使得并发周期非常频繁，大量 GC 线程抢占 CPU，会导致应用程序的性能有所下降。

5.5 回眸：有关对象内存分配和回收的一些细节问题

通过这几个章节的学习，读者应该已经对已有的垃圾回收算法和具体的垃圾回收器及其使用有了一定的了解。本节用来讨论一些有趣的问题，更进一步探讨有关对象在内存中的分配以及回收方式。

5.5.1 禁用 System.gc()

在默认情况下，System.gc()会显式直接触发 Full GC，同时对老年代和新生代进行回收。而一般情况下我们认为，垃圾回收应该是自动进行的，无需手工触发。如果过于频繁地触发垃圾回收对系统性能是没有好处的。因此虚拟机提供了一个参数 DisableExplicitGC 来控制是否手工触发 GC。

System.gc()的实现如下所示：

```
Runtime.getRuntime().gc();
```

Runtime.gc()是一个 native 方法，最终实现在 jvm.cpp 中，如下所示：

```
JVM_ENTRY_NO_ENV(void, JVM_GC(void))
  JVMWrapper("JVM_GC");
  if (!DisableExplicitGC) {
    Universe::heap()->collect(GCCause::_java_lang_system_gc);
  }
JVM_END
```

可以看到，如果设置了-XX:-+DisableExplicitGC，条件判断就无法成立，那么就会禁用显式 GC，使得 System.gc()等价于一个空函数调用。

5.5.2 System.gc()使用并发回收

默认的情况下，即使 System.gc()生效，它会使用传统的 Full GC 方式回收整个堆，而会忽略参数中的 UseG1GC 和 UseConcMarkSweepGC。比如使用以下参数运行程序：

```
-XX:+PrintGCDetails -XX:+UseConcMarkSweepGC
或者
-XX:+PrintGCDetails -XX:+UseG1GC
```

遇到 System.gc()的时候，则会有以下日志输出。

对于 CMS：

```
[Full GC[CMS: 454K->453K(10944K), 0.0046875 secs] 544K->453K(15936K), [CMS Perm : 1593K->1593K(12288K)], 0.0047210 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
```

对于 G1：

```
[Full GC 616K->453K(5120K), 0.0049140 secs]
   [Eden: 1024.0K(7168.0K)->0.0B(2048.0K) Survivors: 0.0B->0.0B Heap: 616.5K(16.0M)->453.4K(5120.0K)]
 [Times: user=0.01 sys=0.00, real=0.00 secs]
```

显然，在此时，CMS 和 G1 是没有并发执行的，因为在日志中没有任何并发相关信息。打开虚拟机参数-XX:+ExplicitGCInvokesConcurrent 后，可以改变这种默认的行为，比如使用以下参数：

```
-XX:+PrintGCDetails -XX:+UseConcMarkSweepGC -XX:+ExplicitGCInvokesConcurrent
或者
-XX:+PrintGCDetails -XX:+UseG1GC -XX:+ExplicitGCInvokesConcurrent
```

那么对于 CMS：

```
[GC[ParNew: 620K->462K(4928K), 0.0012471 secs] 620K->462K(15872K), 0.0012948 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [1 CMS-initial-mark: 0K(10944K)] 462K(15872K), 0.0004039 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[CMS-concurrent-mark: 0.006/0.006 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
[CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC[YG occupancy: 550 K (4928 K)][Rescan (parallel) , 0.0002013 secs][weak refs processing, 0.0000060 secs][scrub string table, 0.0000209 secs] [1 CMS-remark: 0K(10944K)] 550K(15872K), 0.0002639 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[CMS-concurrent-sweep: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

对于 G1（省略部分输出）：

```
[GC pause (young) (initial-mark), 0.0013322 secs]
   [Parallel Time: 1.1 ms, GC Workers: 2]
   [Eden: 1024.0K(7168.0K)->0.0B(5120.0K) Survivors: 0.0B->1024.0K Heap: 616.5K(16.0M)->476.1K(16.0M)]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC concurrent-root-region-scan-start]
[GC concurrent-root-region-scan-end, 0.0003496 secs]
[GC concurrent-mark-start]
[GC concurrent-mark-end, 0.0000331 secs]
[GC remark [GC ref-proc, 0.0000142 secs], 0.0003168 secs]
```

```
 [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC cleanup 517K->517K(16M), 0.0000742 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
```

可以看到，只有在打开 ExplicitGCInvokesConcurrent 开关后，System.gc()这种显式 GC 才会使用并发的方式进行回收，否则，无论是否启用了 CMS 或者 G1，都不会进行并发回收。

5.5.3 并行 GC 前额外触发的新生代 GC

对于并行回收器的 Full GC（使用 UseParallelOldGC 或者 UseParallelGC），细心的读者可能会发现，在每一次 Full GC 之前都会伴随一次新生代 GC。这和串行回收器相比，有很大的不同。

【示例 5-3】下面给出一个简单的比较示例。

```
public class ScavengeBeforeFullGC {
    public static void main(String args[]){
        System.gc();
    }
}
```

上述代码什么也没做，只是进行了一次简单的 Full GC。使用以下参数，设置为串行回收器运行代码：

```
-XX:+PrintGCDetails -XX:+UseSerialGC
```

系统进行的 GC 如下：

```
[Full GC[Tenured: 0K->376K(10944K), 0.0033328 secs] 603K->376K(15872K), [Perm :
142K->142K(12288K)], 0.0033825 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

一切都在意料之中，System.gc()触发了一次 Full GC 操作。而切换为并行回收器，再尝试同样的操作：

```
-XX:+PrintGCDetails  -XX:+UseParallelOldGC
```

输出如下：

```
[GC [PSYoungGen: 670K->480K(5120K)] 670K->480K(15872K), 0.0153729 secs]
[Times: user=0.02 sys=0.00, real=0.02 secs]
[Full GC [PSYoungGen: 480K->0K(5120K)] [ParOldGen: 0K->453K(10752K)]
480K->453K(15872K) [PSPermGen: 1592K->1591K(12288K)], 0.0073608 secs] [Times:
user=0.00 sys=0.00, real=0.01 secs]
```

可以看到，在使用并行回收器时，触发 Full GC 之前，进行了一次新生代 GC。因此，这里的 System.gc()实际上触发了两次 GC。这样做的目的是先将新生代进行一次收集，避免将所有回收工作同时交给一次 Full GC 进行，从而尽可能地缩短一次停顿时间。

如果不需要这个特性，那么可以使用参数-XX:-ScavengeBeforeFullGC 去除发生在 Full GC 之前的那次新生代 GC。默认情况下，ScavengeBeforeFullGC 的值为 true。

5.5.4 对象何时进入老年代

对于一般情况而言，当对象首次创建时，会被放置在新生代的 eden 区。为什么取名叫 eden 呢？因为 eden 区也就是“伊甸园”。根据圣经的记载，亚当和夏娃就住在伊甸园，那也是人类开始居住的地方。这里沿用伊甸园的名字也就是这个意思。在堆中分配的对象首先会被安置在 eden 区。如果没有 GC 的介入，那么这些对象不会离开 eden。

1. 初创的对象在 eden 区

【示例 5-4】下面的代码申请了大约 5MB 内存。

```
public class AllocEden {
    public static final int _1K=1024;
    public static void main(String args[]){
        for(int i=0;i<5*_1K;i++){
            byte[] b=new byte[_1K];
        }
    }
}
```

使用如下参数运行上面的代码：

```
-Xmx64M -Xms64M -XX:+PrintGCDetails
```

得到输出：

```
Heap
 def new generation   total 19648K, used 6321K [0x30460000, 0x319b0000, 0x319b0000)
  eden space 17472K,  36% used [0x30460000, 0x30a8c500, 0x31570000)
  from space 2176K,   0% used [0x31570000, 0x31570000, 0x31790000)
  to   space 2176K,   0% used [0x31790000, 0x31790000, 0x319b0000)
 tenured generation   total 43712K, used 0K [0x319b0000, 0x34460000, 0x34460000)
   the space 43712K,   0% used [0x319b0000, 0x319b0000, 0x319b0200, 0x34460000)
 compacting perm gen  total 12288K, used 142K [0x34460000, 0x35060000, 0x38460000)
   the space 12288K,   1% used [0x34460000, 0x34483a88, 0x34483c00, 0x35060000)
```

```
   ro space 10240K,  44% used [0x38460000, 0x388d73f0, 0x388d7400, 0x38e60000)
 rw space 12288K,  52% used [0x38e60000, 0x394add28, 0x394ade00, 0x39a60000)
```

可以看到整个过程中没有 GC 发生，因此，一共分配的 5MB 数据都应该在堆中，从堆的日志中可以看到，eden 区占据了 6MB 左右的空间，from、to 和老年代 tenured 均未使用。

2. 老年对象进入老年代

那 eden 区中的对象何时能进入老年代呢？一般来说，当对象的年龄达到一定的大小，就自然可以离开年轻代，进入老年代，一般可以把对象进入老年代的事件，称为“晋升”。对象的年龄是由对象经历过的 GC 次数决定的。在新生代中的对象每经历一次 GC，如果它没有被回收，它的年龄就加 1。虚拟机提供了一个参数来控制新生代对象的最大年龄：MaxTenuringThreshold。默认情况下，这个参数为 15。也就是说，在新生代的对象最多经历 15 次 GC，就可以晋升到老年代。

【示例 5-5】修改上述代码，使之如下：

```
01 public class MaxTenuringThreshold {
02     public static final int _1M=1024*1024;
03     public static final int _1K=1024;
04     public static void main(String args[]){
05         Map<Integer,byte[]> map=new HashMap<Integer,byte[]>();
06         for(int i=0;i<5*_1K;i++){
07             byte[] b=new byte[_1K];
08             map.put(i, b);
09         }
10 
11         for(int k=0;k<17;k++){
12             for(int i=0;i<270;i++){
13                 byte[] g=new byte[_1M];
14             }
15         }
16     }
17 }
```

上述代码依然申请了大约 5MB 空间，不同的是，在代码第 8 行，将新生成的 byte 数组进行保存，防止它们在 GC 时被回收。代码第 11~15 行不停地在新生代分配内存，以触发新生代 GC。使用如下参数运行代码：

```
   -Xmx1024M   -Xms1024M   -XX:+PrintGCDetails   -XX:MaxTenuringThreshold=15
-XX:+PrintHeapAtGC
```

这里分配了 1GB 的内存，用意是将对象尽可能预留在新生代（一个大的堆自然有一个大的新生代）。显式指定了 MaxTenuringThreshold 为 15（和默认值一样，这里方便读者理解），并打开了 PrintHeapAtGC 开关，在每次 GC 时，都打印堆的详细信息。

程序的部分输出如下（由于打开了 PrintHeapAtGC，会产生大量信息，这里仅显示相关部分）：

```
{Heap before GC invocations=0 (full 0):
 def new generation   total 314560K, used 279477K [0x04940000, 0x19e90000, 0x19e90000)
  eden space 279616K,  99% used [0x04940000, 0x15a2d770, 0x15a50000)
  from space 34944K,   0% used [0x15a50000, 0x15a50000, 0x17c70000)
  to   space 34944K,   0% used [0x17c70000, 0x17c70000, 0x19e90000)
 tenured generation   total 699072K, used 0K [0x19e90000, 0x44940000, 0x44940000)
   the space 699072K,   0% used [0x19e90000, 0x19e90000, 0x19e90200, 0x44940000)
.....
[GC[DefNew:  279477K->5888K(314560K),  0.0053669  secs]  279477K->5888K(1013632K), 0.0053970 secs] [Times: user=0.00 sys=0.02, real=0.01 secs]
Heap after GC invocations=1 (full 0):
 def new generation   total 314560K, used 5888K [0x04940000, 0x19e90000, 0x19e90000)
  eden space 279616K,   0% used [0x04940000, 0x04940000, 0x15a50000)
  from space 34944K,  16% used [0x17c70000, 0x18230358, 0x19e90000)
  to   space 34944K,   0% used [0x15a50000, 0x15a50000, 0x17c70000)
 tenured generation   total 699072K, used 0K [0x19e90000, 0x44940000, 0x44940000)
   the space 699072K,   0% used [0x19e90000, 0x19e90000, 0x19e90200, 0x44940000)
.....(这里省略中间十几次GC，它们的情况和第一次GC是一样的)
{Heap before GC invocations=15 (full 0):
 def new generation   total 314560K, used 285452K [0x04940000, 0x19e90000, 0x19e90000)
  eden space 279616K,  99% used [0x04940000, 0x15a43190, 0x15a50000)
  from space 34944K,  16% used [0x17c70000, 0x18230250, 0x19e90000)
  to   space 34944K,   0% used [0x15a50000, 0x15a50000, 0x17c70000)
 tenured generation   total 699072K, used 0K [0x19e90000, 0x44940000, 0x44940000)
   the space 699072K,   0% used [0x19e90000, 0x19e90000, 0x19e90200, 0x44940000)
....
[GC[DefNew: 285452K->0K(314560K), 0.0045594 secs] 285452K->5888K(1013632K), 0.0045806 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap after GC invocations=16 (full 0):
 def new generation   total 314560K, used 0K [0x04940000, 0x19e90000, 0x19e90000)
  eden space 279616K,   0% used [0x04940000, 0x04940000, 0x15a50000)
  from space 34944K,   0% used [0x15a50000, 0x15a50100, 0x17c70000)
  to   space 34944K,   0% used [0x17c70000, 0x17c70000, 0x19e90000)
 tenured generation   total 699072K, used 5888K [0x19e90000, 0x44940000, 0x44940000)
   the space 699072K,   0% used [0x19e90000, 0x1a450150, 0x1a450200, 0x44940000)
....
```

观察以上日志，在第一次 GC 开始前，eden 使用了 99%，这也是触发新生代 GC 的原因。既然 eden 区不能容纳更多对象，而后又需要有新的对象产生，那自然需要对 eden 区进行清理，而清理的结果是将存活对象移入了 from。从堆日志中可知，from 区占用了 16%，34944K *0.16=5591KB，大约为 5MB，与放置在 map 对象中的 byte 数量匹配，第一次 GC 的另一个影响是 eden 区被清空。

而后的 14 次 GC 情况和第 1 次是一样的，故在日志中被省略。而每一次 GC 都会使存活对象的年龄加 1，当第 16 次 GC 发生时，可以看到它已将新生代清空。

```
[GC[DefNew: 285452K->0K(314560K), 0.0045594 secs]
```

对比第一次的日志：

```
[GC[DefNew: 279477K->5888K(314560K), 0.0053669 secs]
```

这里有着明显的差异。从新生代被移除的对象，这里晋升到了老年代（指 map 对象中的 byte 数组），这从最后一次 GC 的后续堆日志中可以看到。老年代已经有 5888KB 被使用，而新生代使用为 0KB。这说明这 5MB 对象晋升老年代成功。

读者可以尝试把 MaxTenuringThreshold 改为 10，那么在这个示例中，第 11 次 GC 时，5MB 对象就会晋升到老年代，此处从略。

虽然有上述示例做铺垫，但仍然需要再次强调，MaxTenuringThreshold 指的是最大晋升年龄。它是对象晋升老年代的充分非必要条件。即达到该年龄，对象必然晋升，而未达到该年龄，对象也有可能晋升。事实上，对象的实际晋升年龄，是由虚拟机在运行时自行判断的。

计算晋升年龄的基本逻辑代码如下所示：

```
size_t desired_survivor_size = (size_t)((((double) survivor_capacity)*TargetSurvivorRatio)/100);
size_t total = 0;
int age = 1;
assert(sizes[0] == 0, "no objects with age zero should be recorded");
while (age < table_size) {
  total += sizes[age];
  // check if including objects of age 'age' made us pass the desired
  // size, if so 'age' is the new threshold
  if (total > desired_survivor_size) break;
  age++;
}
int result = age < MaxTenuringThreshold ? age : MaxTenuringThreshold;
```

上述代码中，第 1 行的 desired_survivor_size 定义了期望的 survivor 区的使用大小。在第 5~10 行，根据 desired_survivor_size 计算对象晋升年龄。其中 sizes 数组保存每一个年龄段的对象大小之和。比如 sizes[1]年龄为 1 的，所有对象的大小之和。第 6 行对每一个年龄段的对象大小进行累计求和，如果大于 desired_survivor_size 则退出循环，否则 age（对象晋升年龄）加 1。在对所有年龄段的对象进行统计后，如果发现 survivor 区的幸存对象超过了期望值 desired_survivor_size，那么 age 就成为了晋升年龄。之后，在 age 和 MaxTenuringThreshold 中取较小者作为对象的实际晋升年龄。因此，确定对象何时晋升的另外一个重要参数为 TargetSurvivorRatio，它用于设置 survivor 区的目标使用率，默认为 50，即如果 survivor 区在 GC 后超过 50%的使用率，那么，就很可能会使用较小的 age 作为晋升年龄。

在本示例中，使用如下参数运行代码：

```
-Xmx1024M  -Xms1024M  -XX:+PrintGCDetails  -XX:MaxTenuringThreshold=15
-XX:+PrintHeapAtGC -XX:TargetSurvivorRatio=15
```

将 TargetSurvivorRatio 设置为 15。通过前文的 GC 日志可知，from 区的使用率会维持在 16%，而现在更改目标使用率为 15%，小于 16%，故应该会使对象更快地晋升到老年代。

```
{Heap before GC invocations=1 (full 0):
 def new generation   total 314560K, used 285201K [0x04a10000, 0x19f60000, 0x19f60000)
  eden space 279616K,  99% used [0x04a10000, 0x15ad43e0, 0x15b20000)
  from space 34944K,  16% used [0x17d40000, 0x18300368, 0x19f60000)
  to   space 34944K,   0% used [0x15b20000, 0x15b20000, 0x17d40000)
 tenured generation   total 699072K, used 0K [0x19f60000, 0x44a10000, 0x44a10000)
   the space 699072K,   0% used [0x19f60000, 0x19f60000, 0x19f60200, 0x44a10000)
 ....
 [GC[DefNew: 285201K->0K(314560K), 0.0040121 secs] 285201K->5888K(1013632K), 0.0040337
secs] [Times: user=0.00 sys=0.02, real=0.00 secs]
Heap after GC invocations=2 (full 0):
 def new generation   total 314560K, used 0K [0x04a10000, 0x19f60000, 0x19f60000)
  eden space 279616K,   0% used [0x04a10000, 0x04a10000, 0x15b20000)
  from space 34944K,   0% used [0x15b20000, 0x15b20100, 0x17d40000)
  to   space 34944K,   0% used [0x17d40000, 0x17d40000, 0x19f60000)
 tenured generation   total 699072K, used 5888K [0x19f60000, 0x44a10000, 0x44a10000)
   the space 699072K,   0% used [0x19f60000, 0x1a520158, 0x1a520200, 0x44a10000)
 ....
```

如上日志显示，在第 2 次 GC 时，map 中的 byte 数组对象都已经晋升到老年代。

注意： 对象的实际晋升年龄是根据 survivor 区的使用情况动态计算得来的，而 MaxTenuringThreshold 只是表示这个年龄的最大值。

3. 大对象进入老年代

除了年龄外，对象的体积也会影响对象的晋升。试想，如果对象体积很大，新生代无论 eden 区或者 survivor 区无法容纳这个对象，自然这个对象无法存放在新生代，因此，由于体积太大，也非常有可能被直接晋升到老年代。如图 5.11 所示，如果需要一个连续的 6MB 空间，而新生代 survivor 区（只有 5MB）无法接纳这样的大小，此时无论该对象年龄如何，它都会被直接晋升到老年代。

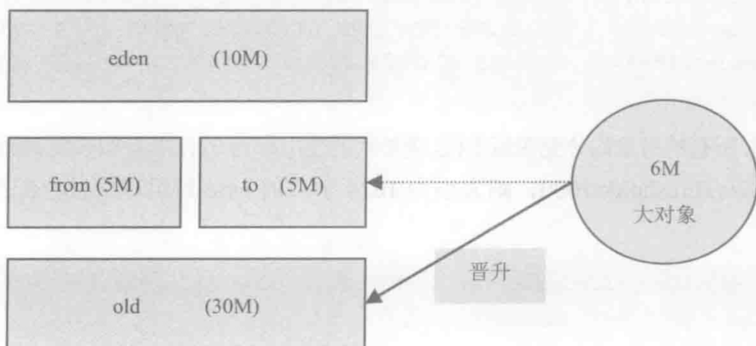


图 5.11　新生代区间无法容纳大对象

另外一个有趣的参数是 PretenureSizeThreshold，它用来设置对象直接晋升到老年代的阈值，单位是字节。只要对象的大于指定值，就会绕过新生代，直接在老年代分配。这个参数只对串行回收器和 ParNew 有效，对于 ParallelGC 无效。默认情况下该值为 0，也就是不指定最大的晋升大小，一切由运行情况决定。

【示例 5-6】下面的示例演示了 PretenureSizeThreshold 的使用。

```
public class PretenureSizeThreshold {
    public static final int _1K=1024;
    public static void main(String args[]){
        Map<Integer,byte[]> map=new HashMap<Integer,byte[]>();
        for(int i=0;i<5*_1K;i++){
            byte[] b=new byte[_1K];
            map.put(i, b);
        }
    }
}
```

上述代码分配了约 5MB 空间，大约为 6 千个 byte 数组，每个数组大小为 1024。使用下面参数运行这段代码：

```
-Xmx32m -Xms32m -XX:+UseSerialGC -XX:+PrintGCDetails
```

得到的部分输出如下：

```
Heap
 def new generation   total 9792K, used 6204K [0x32460000, 0x32f00000, 0x32f00000)
  eden space 8704K,  71% used [0x32460000, 0x32a6f358, 0x32ce0000)
  from space 1088K,   0% used [0x32ce0000, 0x32ce0000, 0x32df0000)
  to   space 1088K,   0% used [0x32df0000, 0x32df0000, 0x32f00000)
 tenured generation   total 21888K, used 0K [0x32f00000, 0x34460000, 0x34460000)
   the space 21888K,   0% used [0x32f00000, 0x32f00000, 0x32f00200, 0x34460000)
....
```

可以看到，所有的对象均分配在新生代，老年代的使用率为 0。接着附加参数 PretenureSizeThreshold，令 PretenureSizeThreshold=1000，则大小为 1024 字节的 byte 数组理应被分配在老年代，参数如下：

```
-Xmx32m -Xms32m -XX:+UseSerialGC -XX:+PrintGCDetails -XX:PretenureSizeThreshold=
1000
```

得到的部分输入日志如下：

```
Heap
 def new generation   total 9792K, used 6124K [0x32460000, 0x32f00000, 0x32f00000)
  eden space 8704K,  70% used [0x32460000, 0x32a5b328, 0x32ce0000)
  from space 1088K,   0% used [0x32ce0000, 0x32ce0000, 0x32df0000)
  to   space 1088K,   0% used [0x32df0000, 0x32df0000, 0x32f00000)
 tenured generation   total 21888K, used 80K [0x32f00000, 0x34460000, 0x34460000)
   the space 21888K,   0% used [0x32f00000, 0x32f14030, 0x32f14200, 0x34460000)
....
```

读者也许会觉得非常奇怪，因为期望的结果是至少有 5MB 数据被分配在老年代，但为什么作为分配主体的 5MB 数组看起来依然在新生代呢？似乎 PretenureSizeThreshold 不起作用，但是老年代的情况却和不加 PretenureSizeThreshold 时有所不同，在这里，有 80KB 的空间被使用。

导致这种现象的原因是虚拟机在为线程分配空间时，会优先使用一块叫作 TLAB 的区域（有关 TLAB 的相关内容在下一节详细介绍），对于体积不大的对象，很有可能会在 TLAB 上先行分配，因此，就失去了在老年代分配的机会。因此，这里简单地禁用 TLAB 即可。使用下述参数，再次运行程序：

```
-Xmx32m -Xms32m -XX:+UseSerialGC -XX:+PrintGCDetails  -XX:-UseTLAB  -XX:
PretenureSizeThreshold=1000
```

得到的部分输出如下：

```
Heap
 def new generation   total 9792K, used 580K [0x32460000, 0x32f00000, 0x32f00000)
  eden space 8704K,   6% used [0x32460000, 0x324f13c0, 0x32ce0000)
  from space 1088K,   0% used [0x32ce0000, 0x32ce0000, 0x32df0000)
  to   space 1088K,   0% used [0x32df0000, 0x32df0000, 0x32f00000)
 tenured generation   total 21888K, used 5413K [0x32f00000, 0x34460000, 0x34460000)
   the space 21888K,  24% used [0x32f00000, 0x33449470, 0x33449600, 0x34460000)
```

可以看到，在禁用 TLAB 后，大于 1000 字节的 byte 数组已经分配在老年代了。那什么是 TLAB 呢？请读者继续阅读下一节内容。

5.5.5　在 TLAB 上分配对象

TLAB 的全称是 Thread Local Allocation Buffer，即线程本地分配缓存。从名字上可以看到，TLAB 是一个线程专用的内存分配区域。

为什么需要 TLAB 这个区域呢？这是为了加速对象分配而生的。由于对象一般会分配在堆上，而堆是全局共享的。因此在同一时间，可能会有多个线程在堆上申请空间。因此，每一次对象分配都必须要进行同步，而在竞争激烈的场合分配的效率又会进一步下降。考虑到对象分配几乎是 Java 最常用的操作，因此 Java 虚拟机就使用了 TLAB 这种线程专属的区间来避免多线程冲突，提高了对象分配的效率。TLAB 本身占用了 eden 区的空间。在 TLAB 启用的情况下，虚拟机会为每一个 Java 线程分配一块 TLAB 空间。

【示例 5-7】下面来看一下启用 TLAB 与关闭 TLAB 时的性能差异。

```
public class UseTLAB {
    public static void alloc(){
        byte[] b=new byte[2];
        b[0]=1;

    }
    public static void main(String args[]){
        long b=System.currentTimeMillis();
        for(int i=0;i<10000000;i++){
            alloc();
        }
        long e=System.currentTimeMillis();
        System.out.println(e-b);
    }
}
```

上述代码进行若干次数组分配，并统计了执行时间。使用如下参数执行：

```
-XX:+UseTLAB -Xcomp -XX:-BackgroundCompilation -XX:-DoEscapeAnalysis -server
```

该参数打开了 TLAB（默认开启，这里为了让表述更清晰），并启用了对所有函数的 JIT 以及禁止后台编译（这里只是希望在相对一致的环境中测试），同时禁用了逃逸分析，以防止栈上分配的行为影响本次测试结果，开启 Server 模式是因为在 Client 模式下，不支持逃逸分析参数 DoEscapeAnalysis。最终程序输出：

```
54
```

修改 Java 虚拟机参数为：

```
-XX:-UseTLAB -Xcomp -XX:-BackgroundCompilation -XX:-DoEscapeAnalysis -server
```

禁用了 TLAB，结果程序输出为：

```
141
```

可以看到，TLAB 是否启用，对于对象分配的影响是很大的。

由于 TLAB 空间一般不会太大，因此大对象无法在 TLAB 上进行分配，总是会直接分配在堆上。TLAB 空间由于比较小，因此很容易装满。比如，一个 100KB 的空间，如果已经使用了 80KB，当需要再分配一个 30KB 的对象时，肯定就无能为力了。这时，虚拟机有两种选择，第一，废弃当前的 TLAB，这样就会浪费 20KB 空间。第二，将这 30KB 的对象直接分配在堆上，保留当前的 TLAB，这样可以希望将来有小于 20KB 的对象分配请求可以直接使用这块空间。当发生请求分配的对象大于 TLAB 内可用空间时，虚拟机如何在这两种行为间抉择呢？虚拟机内部会维护一个叫作 refill_waste 的值，当请求对象大于 refill_waste 时，会选择在堆中分配，若小于该值，则会废弃当前 TLAB，新建 TLAB 来分配新对象。这个阈值可以使用虚拟机参数 TLABRefillWasteFraction 来调整，它表示 TLAB 中允许产生这种浪费的比例。默认值为 64，即表示使用约为 1/64 的 TLAB 空间大小作为 refill_waste。

默认情况下，TLAB 和 refill_waste 都是会在运行时不断调整的，使系统的运行状态达到最优。如果想要禁用自动调整 TLAB 的大小，可以使用-XX:-ResizeTLAB 禁用 ResizeTLAB，并使用-XX:TLABSize 手工指定一个 TLAB 的大小。

如果想观察 TLAB 的使用情况，则打开跟踪参数-XX:+PrintTLAB。现使用如下参数再次运行这段代码：

```
-XX:+UseTLAB -XX:+PrintTLAB -XX:+PrintGC -XX:TLABSize=102400 -XX:-ResizeTLAB
-XX:TLABRefillWasteFraction=100 -XX:-DoEscapeAnalysis -server
```

可以得到如下输出（篇幅有限，只截取部分输出）：

```
    TLAB: gc thread: 0x4c623400 [id: 1932] desired_size: 100KB slow allocs: 0
refill waste: 1024B alloc: 0.29593     5000KB refills: 1 waste 100.0% gc: 102368B
slow: 0B fast: 0B
    TLAB: gc thread: 0x4c621000 [id: 1740] desired_size: 100KB slow allocs: 0
refill waste: 1024B alloc: 0.29593     5000KB refills: 1 waste 100.0% gc: 102400B
slow: 0B fast: 0B
    TLAB: gc thread: 0x0045c400 [id: 5168] desired_size: 100KB slow allocs: 0
refill waste: 1024B alloc: 0.64808    10950KB refills: 167 waste  0.0% gc: 0B slow:
2672B fast: 0B
    TLAB totals: thrds: 3  refills: 169 max: 167 slow allocs: 0 max 0 waste:  1.2%
gc: 204768B max: 102400B slow: 2672B max: 2672B fast: 0B max: 0B
    [GC 17392K->472K(79872K), 0.0027370 secs]
    ….
    90
```

上述日志就是 TLAB 的使用日志，分为两部分，首先是每一个线程的 TLAB 的使用情况，其次是以 TLAB totals 为首的整体 TLAB 的统计情况。

在输出日志中，desired_size 为 TLAB 的大小，这里通过-XX:TLABSize=102400 指定为 100KB，slow allocs 表示从上一次新生代 GC 到现在为止慢分配次数，这里的慢分配是指由于 TLAB 空闲空间太小不能满足较大对象的分配，而将对象直接分配到堆上。后面的 refill waste 表示前文所述的 refill_waste 值。后续的 alloc 表示当前线程的 TLAB 分配比例和使用评估量，这是一个统计数据，前者表示自上一次新生代 GC 后的 number_of_refills * desired_size/used_tlab 的加权平均值，后者为该平均值乘以 used_tlab（意为这个 TLAB 上大约合计被分配了多少空间）。日志中后续的 refills 表示该线程的 TLAB 空间被重新分配并填充的次数，waste 表示空间的浪费比例。

浪费的空间由三部分组成，即后续的 gc、slow 和 fast。其中，gc 表示在当前新生代 GC 发生时，尚空闲的 TLAB 空间，slow 和 fast 都表示当 TLAB 被废弃时尚未被使用的 TLAB 空间，两者的不同是 fast 表示这个 refill 操作是通过 JIT 编译优化的（禁用 JIT，那么 fast 永远为 0）。而 wast 比例则是由浪费空间之和（gc+slow+fast）与总分配大小（_number_of_refills * _desired_size）的比值。

最后的 TLAB totals 则显示了所有线程的统计情况。如 thrds 显示了相关线程总数，refills 表示所有线程 refills 的总数，紧跟在 refills 后的 max，表示 refills 次数最多的线程的 refills 次数。

图 5.12 展示了简要的对象分配流程。首先，如果运行栈上分配，系统会先进行栈上分配，

没有开启栈上分配或者不符合条件则会进行 TLAB 分配，如果 TLAB 分配不成功，再尝试在堆上分配，如果满足了直接进入老年代的条件（PretenureSizeThreshold 等参数），就在老年代分配，否则就在 eden 区分配对象，当然，如果有必要，可能会进行一次新生代 GC。

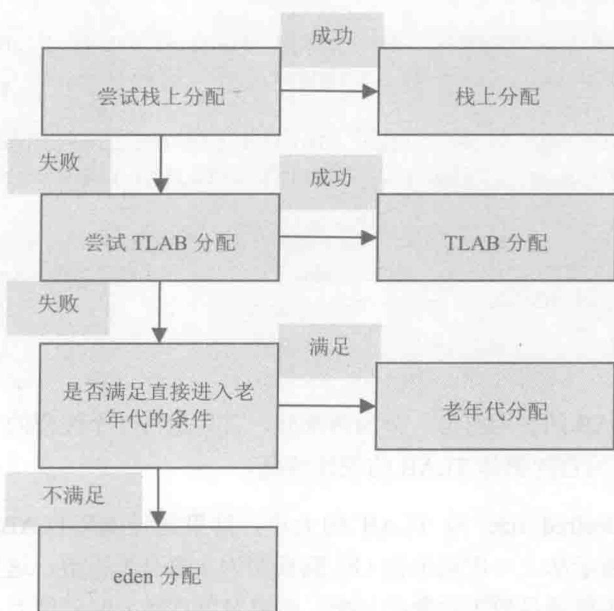


图 5.12　对象分配简要流程

5.5.6　方法 finalize()对垃圾回收的影响

Java 中提供了一个类似于 C++中析构函数的机制——finalize()函数，它在 java.lang.Object 中被申明，其形式如下：

```
protected void finalize() throws Throwable { }
```

该函数允许在子类中被重载，用于在对象被回收时进行资源地释放。目前，普遍的认识是，尽量不要使用 finalize()函数进行资源释放，原因主要有以下几点：

- 第 4 章曾提到，在 finalize()时可能会导致对象复活；
- finalize()函数的执行时间是没有保障的，它完全由 GC 线程决定，极端情况下，若不发生 GC，则 finalize()将没有机会执行；
- 一个糟糕的 finalize()会严重影响 GC 的性能。

函数 finalize()是由 FinalizerThread 线程处理的。每一个即将被回收的并且包含有 finalize()

方法的对象都会在正式回收前加入 FinalizerThread 的执行队列，该队列为 java.lang.ref.ReferenceQueue 引用队列，内部实现为链表结构，队列中每一项为 java.lang.ref.Finalizer 引用对象，它本质为一个引用，如图 5.13 所示，这和虚引用、弱引用等如出一辙。

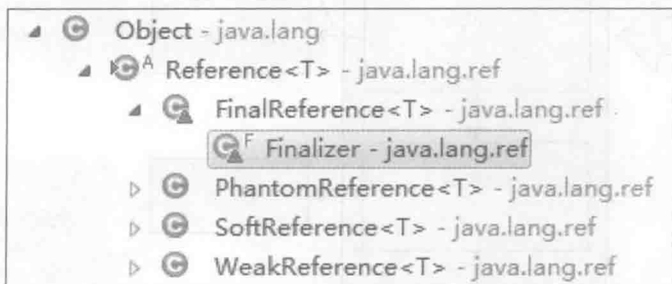


图 5.13　java.lang.ref.Finalizer 引用

Finalizer 内部封装了实际的回收对象，如图 5.14 所示。可以看到 next、prev 为实现链表所需，它们分别指向队列中的下一个元素和上一个元素，而 referent 字段则指向实际的对象引用。比如，在后续的示例中就为 LongFinalize$LF。

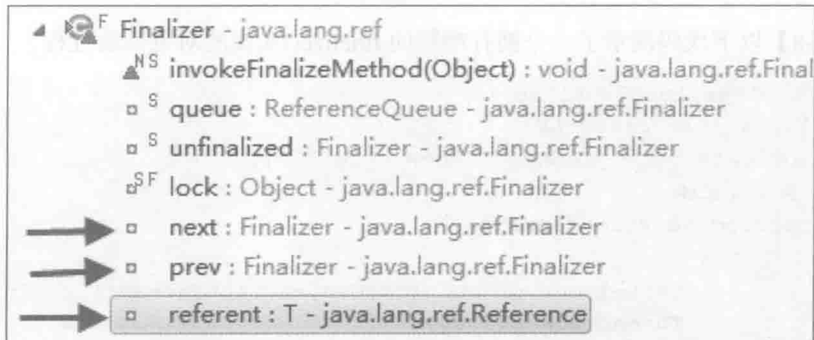


图 5.14　Finalizer 主要结构

由于对象在回收前被 Finalizer 的 referent 字段进行“强引用”，并加入了 FinalizerThread 的执行队列，这意味着对象又变为可达对象，因此阻止了对象的正常回收。由于在引用队列中的元素排队执行 finalize()方法，一旦出现性能问题，将导致这些垃圾对象长时间堆积在内存中，可能会导致 OOM 异常。

图 5.15 显示了 FinalizerThread 的工作过程和 FinalizerThread 执行队列中 Finalizer 的引用关系。

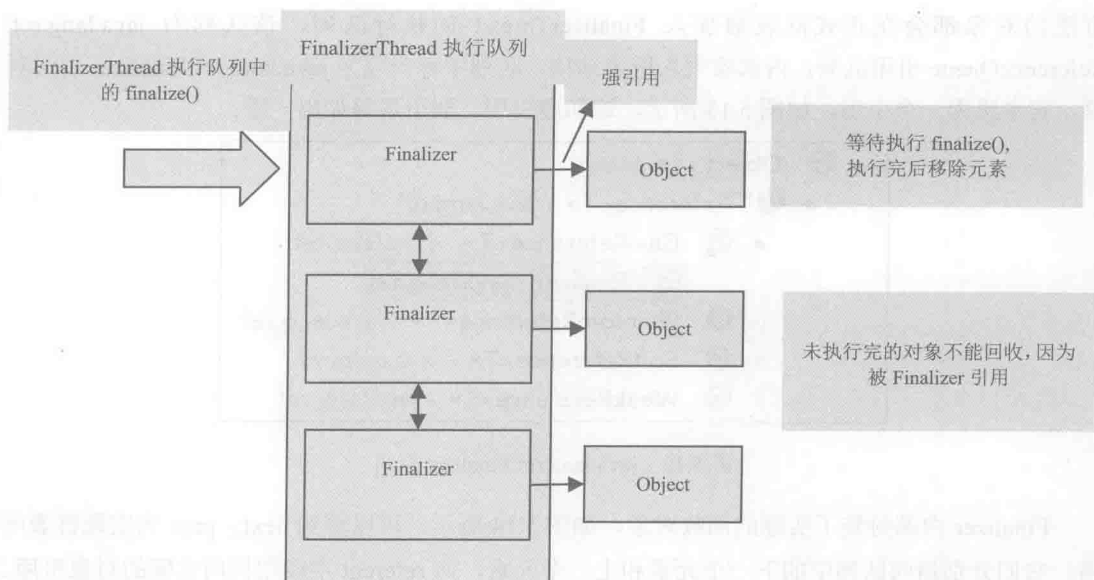


图 5.15 FinalizerThread 执行队列

【示例 5-8】以下代码演示了一个拥有糟糕的 finalize()实现的对象回收过程。

```
public class LongFinalize {
  public static class LF {
      private byte[] content =new byte[512];
      @Override
      protected void finalize(){
          try {
              System.out.println(Thread.currentThread().getId());
              Thread.sleep(1000);
          } catch (Exception e) {
              e.printStackTrace();
          }
      }
  }

  public static void main(String[] args) {
      long b=System.currentTimeMillis();
      for(int i=0;i<50000;i++){
          LF f=new LF();
      }
```

```
20        long e=System.currentTimeMillis();
21        System.out.println(e-b);
22    }
23 }
```

上述代码为一个拥有糟糕 finalize()实现的类 LongFinalize$LF（这是一个内部类，因此带有$符号），可以看到在第 8 行，一个 sleep()方法模拟了一个耗时操作。主函数则不断产生新的 LF 对象。使用如下参数执行上述代码：

```
-Xmx10m -Xms10m -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError -XX:
HeapDumpPath="D:/f.dump"
```

结果发生了 OOM 错误并在 D 盘下得到了堆的 Dump 文件。仔细阅读代码第 16~21 行，不难发现，每次循环中产生的 LF 对象（占用大约 512 字节）都会在下一次循环中失效（因为局部变量作用域过期，对象也无其他引用），因此所有产生的 LF 对象都应该可以被回收。因此 10MB 堆空间，理论上应该完全可以满足需要，只是需要多进行几次 GC 而已，而这里为什么依然会出现 OOM 呢？使用 MAT（有关 MAT 的具体介绍请参考第 7 章）打开得到的堆文件，如图 5.16 所示。

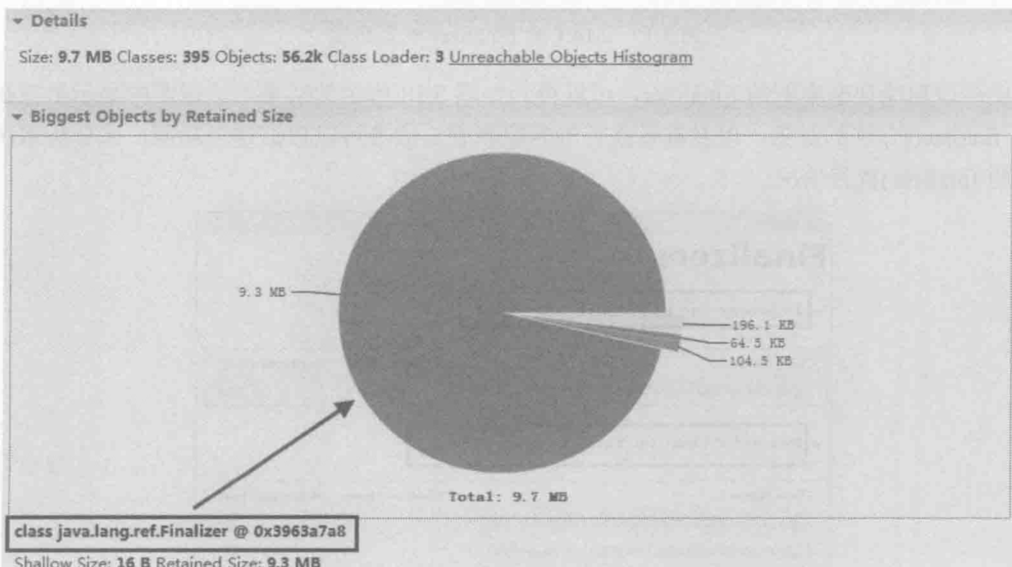


图 5.16　OOM 时显示大量 Finalizer

从最大对象中可以看到，目前系统中有大量的 Finalizer 类，这意味着 FinalizerThread 执行队列可能一直持有对象而来不及执行，因此大量的对象堆积而无法被释放，最终导致了这个

OOM。使用 MAT 自带的"Finalizer Overview"功能可以更好地观察系统中的 Finalizer，如图 5.17 所示。

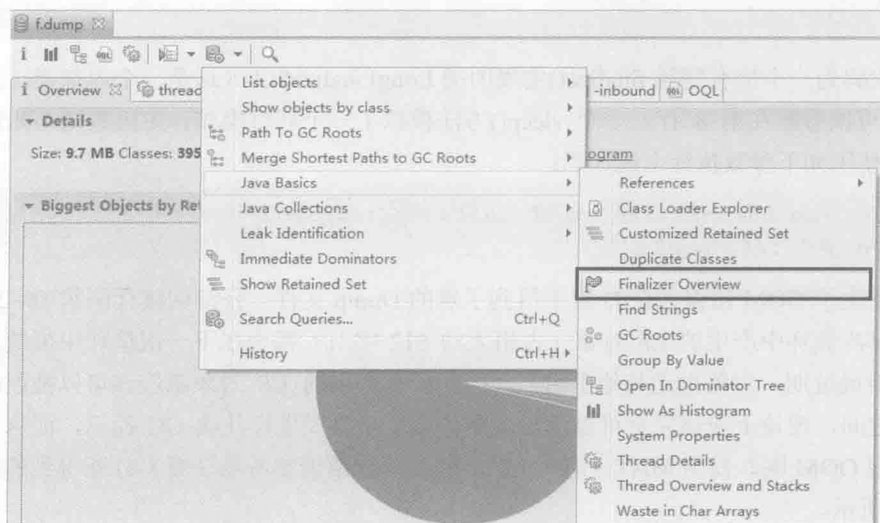


图 5.17　MAT 的 Finalizer 概览功能

使用该功能观察本例的 Finalizer，可以得到如图 5.18 所示的结果。该视图中显示了正在被执行 finalize()方法的对象，以及准备执行的对象列表。读者可以根据这个功能，来分析系统中对象的 finalize()执行情况。

Finalizers

Finalizers

In processing by Finalizer Thread

Class Name	Shallow Heap	Retained Heap
geym.zbase.ch5.LongFinalize$LF @ 0x342040f8 »	16	544

Ready for Finalizer Thread - Object List

Class Name	Shallow Heap	Retained Heap
geym.zbase.ch5.LongFinalize$LF @ 0x33fdeb30 »	16	544
geym.zbase.ch5.LongFinalize$LF @ 0x33c80210 »	16	544
geym.zbase.ch5.LongFinalize$LF @ 0x33fded70 »	16	544
geym.zbase.ch5.LongFinalize$LF @ 0x33c80450 »	16	544
geym.zbase.ch5.LongFinalize$LF @ 0x33fdefb0 »	16	544
geym.zbase.ch5.LongFinalize$LF @ 0x33c80690 »	16	544

图 5.18　FinalizerThread 执行队列的展示

去掉 LF 类的 finalize()方法，即注释本例代码中的第 5~12 行，再次以相同的参数运行这段程序。可以观察到，程序很快正常结束。由此，可以进一步说明 finalize()对 GC 产生的影响。

注意：一个糟糕的 finalize()可能会使对象长时间被 Finalizer 引用，而得不到释放，因此这会进一步增加 GC 的压力。因此，finalize()应该是尽量少地被使用。

虽然 finalize()不推荐使用，但是在有些场合，使用 finalize()可以起到双保险的作用。比如在 MySQL 的 JDBC 驱动中，com.mysql.jdbc.ConnectionImpl 就实现了 finalize()方法，实现代码如下：

```
protected void finalize() throws Throwable {
    cleanup(null);
    super.finalize();
}
```

也就是，当一个 JDBC Connection 被回收时，需要进行连接的关闭，即这里的 cleanup()方法。事实上，在回收前，开发人员如果正常调用了 Connection.close()方法，那么连接就会被显式关闭，那样的话，在 cleanup()方法中将什么都不会做。而如果开发人员忘记显式关闭连接，而 Connection 对象又被回收了，则会隐式地进行连接的关闭，确保没有数据库连接的泄漏。而一般来说，官方总是极其鼓励开发人员在开发过程中显式关闭数据库连接的。因此，finalize()方法可能会被作为一种补偿措施，在正常方法出现意外时（开发人员疏忽）进行补偿，尽可能确保系统稳定。当然，由于其调用时间的不确定性，这不能单独作为可靠的资源回收手段。

5.6 温故又知新：常用的 GC 参数

1. 与串行回收器相关的参数

- -XX:+UseSerialGC：在新生代和老年代使用串行收集器。
- -XX:SurvivorRatio：设置 eden 区大小和 survivior 区大小的比例。
- -XX:PretenureSizeThreshold：设置大对象直接进入老年代的阈值。当对象的大小超过这个值时，将直接在老年代分配。
- -XX:MaxTenuringThreshold：设置对象进入老年代的年龄的最大值。每一次 Minor GC 后，对象年龄就加 1。任何大于这个年龄的对象，一定会进入老年代。

2. 与并行 GC 相关的参数

- -XX:+UseParNewGC：在新生代使用并行收集器。
- -XX:+UseParallelOldGC：老年代使用并行回收收集器。

- -XX:ParallelGCThreads：设置用于垃圾回收的线程数。通过情况下可以和 CPU 数量相等，但在 CPU 数量比较多的情况下，设置相对较小的数值也是合理的。
- -XX:MaxGCPauseMillis：设置最大垃圾收集停顿时间。它的值是一个大于 0 的整数。收集器在工作时，会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMillis 以内。
- -XX:GCTimeRatio：设置吞吐量大小。它的值是一个 0 到 100 之间的整数。假设 GCTimeRatio 的值为 *n*，那么系统将花费不超过 1/(1+*n*)的时间用于垃圾收集。
- -XX:+UseAdaptiveSizePolicy：打开自适应 GC 策略。在这种模式下，新生代的大小、eden 和 survivior 的比例、晋升老年代的对象年龄等参数会被自动调整，以达到在堆大小、吞吐量和停顿时间之间的平衡点。

3. 与 CMS 回收器相关的参数

- -XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用 CMS+串行收集器。
- -XX:ParallelCMSThreads：设定 CMS 的线程数量。
- -XX:CMSInitiatingOccupancyFraction：设置 CMS 收集器在老年代空间被使用多少后触发，默认为 68%。
- -XX:+UseCMSCompactAtFullCollection：设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片的整理。
- -XX:CMSFullGCsBeforeCompaction：设定进行多少次 CMS 垃圾回收后，进行一次内存压缩。
- -XX:+CMSClassUnloadingEnabled：允许对类元数据区进行回收。
- -XX:CMSInitiatingPermOccupancyFraction：当永久区占用率达到这一百分比时，启动 CMS 回收（前提是-XX:+CMSClassUnloadingEnabled 激活了）。
- -XX:UseCMSInitiatingOccupancyOnly：表示只在到达阈值的时候才进行 CMS 回收。
- -XX:+CMSIncrementalMode：使用增量模式，比较适合单 CPU。增量模式在 JDK 8 中标记为废弃，并且将在 JDK 9 中彻底移除。

4. 与 G1 回收器相关的参数

- -XX:+UseG1GC：使用 G1 回收器。
- -XX:MaxGCPauseMillis：设置最大垃圾收集停顿时间。
- -XX:GCPauseIntervalMillis：设置停顿间隔时间。

5. TLAB 相关

- -XX:+UseTLAB：开启 TLAB 分配。

- -XX:+PrintTLAB：打印 TLAB 相关分配信息。
- -XX:TLABSize：设置 TLAB 大小。
- -XX:+ResizeTLAB：自动调整 TLAB 大小。

6. 其他参数

- -XX:+DisableExplicitGC：禁用显式 GC。
- -XX:+ExplicitGCInvokesConcurrent：使用并发方式处理显式 GC。

5.7　动手才是真英雄：垃圾回收器对 Tomcat 性能影响的实验

选择不同的垃圾回收器和堆大小对 Java 应用程序的性能有一定的影响。为了提高系统的性能，应该配置一些合理的虚拟机参数，比如堆大小、垃圾回收器类型等。系统的性能由很多方面决定，比如程序代码的质量、硬件的性能以及网络带宽的延迟等等，当然，虚拟机参数也是中间重要的一环，是保障系统性能的必要非充分条件。也就是说，如果系统性能不令人满意，仅仅调整虚拟机参数也许根本无法解决问题，一个糟糕的系统实现是没有办法用虚拟机参数来弥补的，但如果一个良好的系统实现加上一个糟糕的参数配置，那么系统性能也不会有令人满意的表现。

5.7.1　配置实验环境

本节实验将使用不同的虚拟机参数启动 Tomcat 服务器，通过压力测试，获得虚拟机的主要性能指标，体验不同参数对系统性能的影响。实验背景如下：

- 环境
  - Tomcat 7
  - 一个 JSP 网站
  - 测试网站的吞吐量
- 工具
  - Apache JMeter
- 实验原理
  - 通过 JMeter 对 Tomcat 增加压力，不同的虚拟机参数应该会有不同的性能表现
- 目的
  - 观察不同参数配置对吞吐量的影响

系统结构如图 5.19 所示，为防止 JMeter 对 Tomcat 产生影响，测试时使用两台独立的计算机，通过局域网相连。



图 5.19 实验结构图

5.7.2 配置进行性能测试的工具 JMeter

JMeter 是 Apache 下基于 Java 的一款性能测试和压力测试工具。它基于 Java 开发，可对 HTTP 服务器和 FTP 服务器，甚至是数据库进行压力测试。作为一款专业的压力测试工具，JMeter 功能强大，本节仅简要介绍与本次实验相关的功能。

JMeter 的运行主界面如图 5.20 所示，下面开始详细讲述设置步骤。

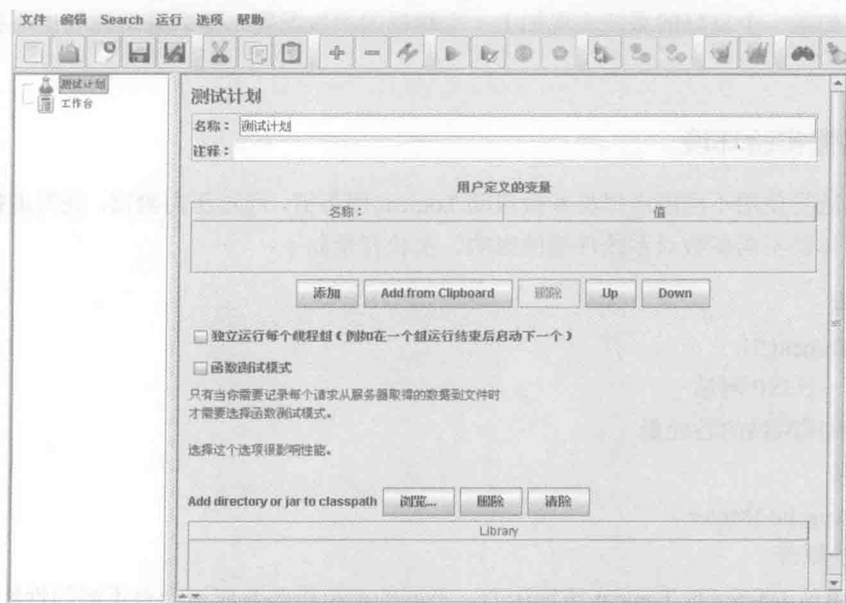


图 5.20 JMeter 运行主界面

（1）为了能够使用 JMeter 对 Tomcat 服务器进行性能测试，首先需要添加线程组，线程组

将模拟用户线程访问 Tomcat 服务器。图 5.21 显示了如何在 JMeter 中添加线程组。

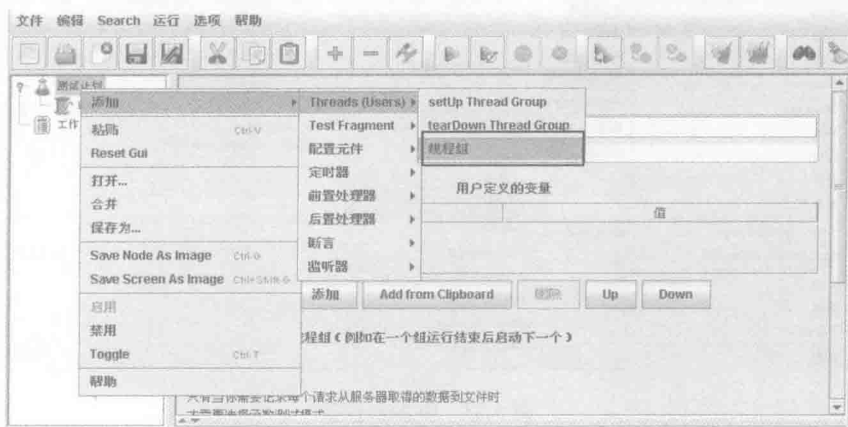


图 5.21　JMeter 中添加线程组

（2）使用线程组可以设置请求的强度，如图 5.22 所示，设置了 10 个线程，并且规定每个线程进行 1000 次请求。这样，Tomcat 就会在这次线程组的运行中，收到 10000 次请求。



图 5.22　JMeter 中配置线程组

（3）除了线程组外，要让 JMeter 正常工作还需要一个采样器。采样器用于对具体的请求进行性能数据的采样，如图 5.23 所示。本例中，需要添加的是 HTTP 请求的采样。

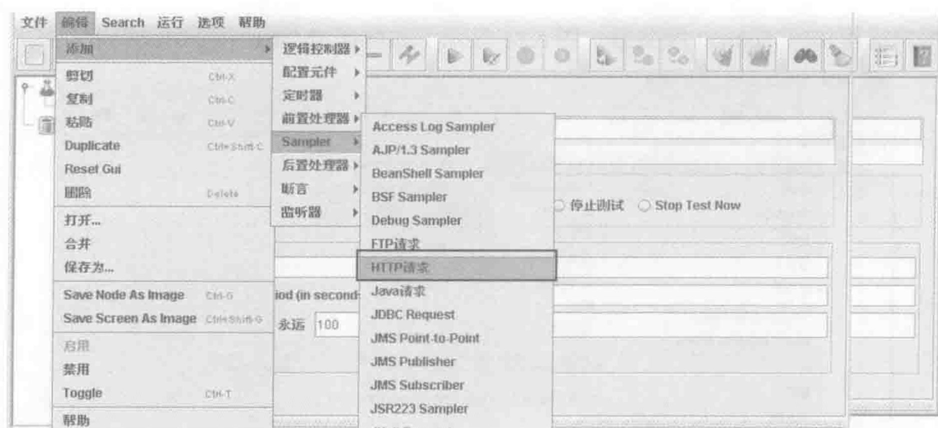


图 5.23　添加采样器

（4）对于添加的 HTTP 请求，还需要对请求的具体目标进行设置，比如，目标服务器的地址、端口号、访问路径等信息，如图 5.24 所示。JMeter 就会按照设置的要求进行批量的请求访问。

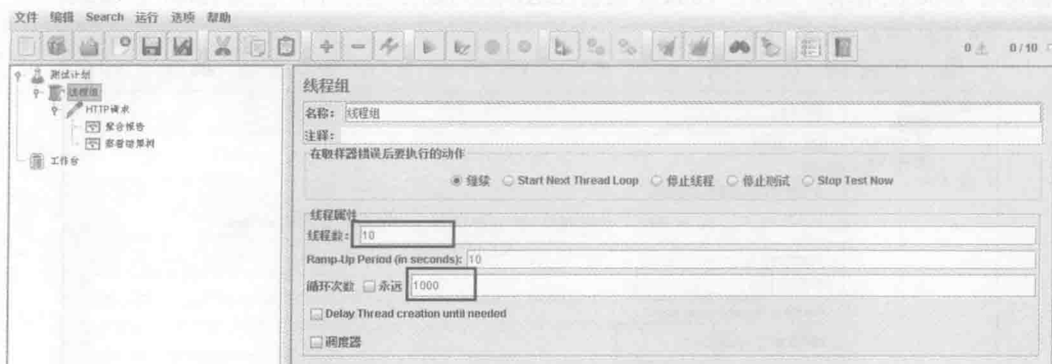


图 5.24　设置 HTTP 请求

（5）对于批量请求的访问结果，JMeter 可以以报表的形式呈现出来。在监听器中，添加聚合报表，如图 5.25 所示。聚合报表可以统计整个测试过程的性能参数。

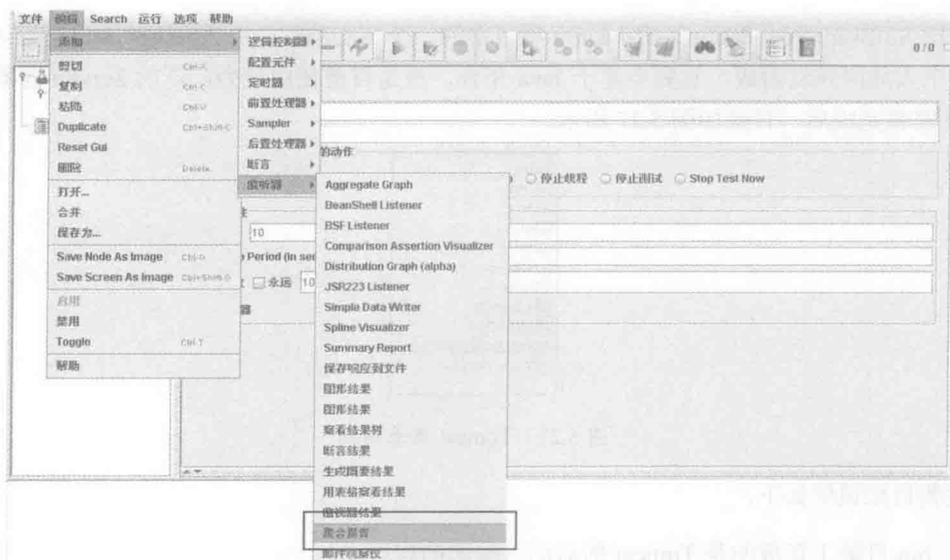


图 5.25　添加聚合报表

（6）添加后，聚合报表的内容如图 5.26 所示。报告中主要内容为每次请求的延时情况和吞吐量。这里主要关注吞吐量，图中用黑色矩形标注部分。

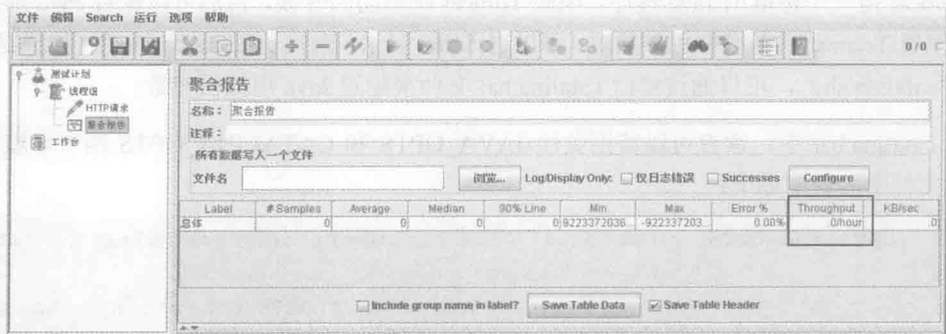


图 5.26　聚合报表呈现的内容

至此，已经完整介绍了通过 JMeter 建立线程组、采样器和聚合报表来进行压力测试，并获得测试结果的方法，本实验将使用这个方法获得 Tomcat 的测试结果。

5.7.3　配置 Web 应用服务器 Tomcat

Tomcat 服务器是一个免费的开放源代码 Web 应用服务器。Tomcat 是 Apache 软件基金会

（Apache Software Foundation）的 Jakarta 项目中的一个核心项目，由 Apache、Sun 和其他一些公司及个人共同开发而成。它完全基于 Java 平台，也是目前使用最为广泛的 Servlet 容器之一。Tomcat 安装完成后，目录如图 5.27 所示。

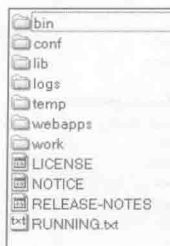


图 5.27　Tomcat 目录结构

主要目录说明如下：

- bin 目录下存放的是 Tomcat 的启动、关闭的程序。
- conf 目录下是 Tomcat 服务器的配置文件，包括 Tomcat 的权限管理、线程池、端口号等配置信息。
- webapps 下存放部署在 Tomcat 下的 web 应用，每个应用对应一个文件夹或者 war 包。

Tomcat 是一个标准的 Java 程序，因此 Tomcat 在启动的时候，自然可以配置 Java 虚拟机的参数。配置 Tomcat 启动虚拟机参数时，可以进入 bin 目录，查找 catalina.bat 文件（如果是 Linux 平台是 catalina.sh），可以通过修改 catalina.bat 文件来配置 Java 虚拟机参数。

在 catalina.bat 中，读者可以特别关注 JAVA_OPTS 和 CATALINA_OPTS 两个环境变量。它们在文件中的说明如下：

```
rem   CATALINA_OPTS   (Optional) Java runtime options used when the "start",
rem                   "run" or "debug" command is executed.
rem                   Include here and not in JAVA_OPTS all options, that should
rem                   only be used by Tomcat itself, not by the stop process,
rem                   the version command etc.
rem                   Examples are heap size, GC logging, JMX ports etc.
rem   JAVA_OPTS        (Optional) Java runtime options used when any command
rem                   is executed.
rem                   Include here and not in CATALINA_OPTS all options, that
rem                   should be used by Tomcat and also by the stop process,
rem                   the version command etc.
rem                   Most options should go into CATALINA_OPTS.
```

这两个环境变量都可以用来控制 Tomcat 启动时的虚拟机参数。CATALINA_OPTS 用于控制 Tomcat 本身的虚拟机参数，比如堆大小、GC 日志等，这个配置不会被 Tomcat 的其他进程（比如 shutdown.bat）使用。JAVA_OPTS 变量的使用范围更广，除了 Tomcat 外，其他相关进程也会使用 JAVA_OPTS 的配置。我们在大部分情况下使用 CATALINA_OPTS 即可。

在本次实验中，均通过设置 CATALINA_OPTS 来控制虚拟机的行为。

5.7.4 实战案例 1——初试串行回收器

使用 JDK1.6 开启 Tomcat，设置参数如下：

```
set CATALINA_OPTS= -Xloggc:gc.log
-XX:+PrintGCDetails -Xmx32M -Xms32M
-XX:+HeapDumpOnOutOfMemoryError -XX:+UseSerialGC -XX:PermSize=32M
```

此参数中，设置最大 32MB 堆。在进行一段时间的请求后，系统部分 GC 日志如图 5.28 所示。

```
32.672: [Full GC 32.672: [Tenured: 21887K->21887K(21888K), 0.0846281 secs] 31675K->29370K(31680K), [Perm : 17566K->17566K(32768K)], 0.0846755 secs]
32.917: [Full GC 32.917: [Tenured: 21887K->21887K(21888K), 0.0836645 secs] 31679K->29472K(31680K), [Perm : 17566K->17566K(32768K)], 0.0837059 secs]
33.150: [Full GC 33.150: [Tenured: 21887K->21887K(21888K), 0.0831402 secs] 31679K->29571K(31680K), [Perm : 17566K->17566K(32768K)], 0.0831883 secs]
33.486: [Full GC 33.486: [Tenured: 21887K->21887K(21888K), 0.0831823 secs] 31675K->29663K(31680K), [Perm : 17566K->17566K(32768K)], 0.0832323 secs]
33.834: [Full GC 33.834: [Tenured: 21888K->21888K(21888K), 0.0855434 secs] 31680K->29752K(31680K), [Perm : 17566K->17566K(32768K)], 0.0855904 secs]
```

图 5.28　实战案例 1 部分 GC 日志

可以看到，Tomcat 进行了大量频繁的 Full GC。老年代大小为 21888KB，并基本用完，整个堆可用大小为 31680KB，也基本使用殆尽。使用 JMeter 观察到的聚合报告如图 5.29 所示。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	6	4	7	2	135	0.00%	540.6/sec	4127.8
总体	10000	6	4	7	2	135	0.00%	540.6/sec	4127.8

图 5.29　实战案例 1 聚合报告

可以看到，在这 10000 次请求中最终的吞吐量定位在 540 次/秒。

5.7.5 实战案例 2——扩大堆以提升系统性能

修改实战案例 1 中的最大堆大小，将其调整为 512MB，整理参数如下：

```
set CATALINA_OPTS=-Xmx512m
-XX:MaxPermSize=32M
-Xloggc:gc.log -XX:+PrintGCDetails
```

使用相同的方式测试新的参数，得到的 GC 日志如图 5.30 所示。

```
0.192: [GC 0.192: [DefNew: 4416K->512K(4928K),      0.0042742 secs] 4416K->779K(15872K),    0.0043156 secs]
0.290: [GC 0.290: [DefNew: 4928K->512K(4928K),      0.0029843 secs] 5195K->1325K(15872K),   0.0030218 secs]
0.391: [GC 0.392: [DefNew: 4928K->512K(4928K),      0.0042914 secs] 5741K->1920K(15872K),   0.0043301 secs]
0.494: [GC 0.494: [DefNew: 4928K->459K(4928K),      0.0033691 secs] 6336K->2305K(15872K),   0.0034062 secs]

38.433: [GC 38.433: [DefNew: 16768K->1573K(18816K), 0.0197601 secs] 41751K->26557K(60456K), 0.0198062 secs]
39.142: [GC 39.142: [DefNew: 18341K->755K(18816K),  0.0064965 secs] 43325K->27302K(60456K), 0.0065439 secs]
39.857: [GC 39.857: [DefNew: 17523K->1508K(18816K), 0.0039589 secs] 44070K->28055K(60456K), 0.0040043 secs]
40.642: [GC 40.642: [DefNew: 18276K->1509K(18816K), 0.0057905 secs] 44823K->28802K(60456K), 0.0058336 secs]
41.576: [GC 41.576: [DefNew: 18277K->1501K(18816K), 0.0055273 secs] 45570K->29543K(60456K), 0.0055726 secs]
```

图 5.30　实战案例 2 部分 GC 日志

可以看到，当设置最大堆为 512MB 后（未设置初始堆），相同启动初期，可用堆约为 16MB，随着请求的增加，最终堆大小扩展到 60MB 左右。同时在日志最后也未出现 Full GC 日志。说明所有的回收都在新生代完成，这是和实验 1 最大的不同。同时，由于新生代 GC 速度普遍比 Full GC 更快，因此，就单次 GC 耗时上也相差约 1 个数量级。

最终，JMeter 的聚合报告如图 5.31 所示。可以看到，吞吐量上升到了 651 次/秒。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	4	4	6	2	141	0.00%	651.2/sec	4972.1
总体	10000	4	4	6	2	141	0.00%	651.2/sec	4972.1

图 5.31　实战案例 2 聚合报告

5.7.6　实战案例 3——调整初始堆大小

根据实战案例 2 的结果可知，系统最终会稳定在 60MB 的堆附近，因此小于 60MB 的堆很有可能会引起大量 GC。因此在实战案例 3 中扩大初始堆为 60MB，设置参数如下：

```
set CATALINA_OPTS=-Xmx512m -Xms64m
-XX:MaxPermSize=32M
-Xloggc:gc.log -XX:+PrintGCDetails
```

启动 Tomcat，并运行 JMeter，得到 Tomcat 的 GC 日志如图 5.32 所示。

```
48.778: [GC 48.778: [DefNew: 19119K->1578K(19712K), 0.0056431 secs] 47170K->30414K(63424K), 0.0056928 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
49.409: [GC 49.409: [DefNew: 19114K->1575K(19712K), 0.0053110 secs] 47950K->31197K(63424K), 0.0053534 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
50.171: [GC 50.171: [DefNew: 19104K->1576K(19712K), 0.0055263 secs] 48727K->31979K(63424K), 0.0055693 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
51.096: [GC 51.096: [DefNew: 19112K->1576K(19712K), 0.0051372 secs] 49515K->32762K(63424K), 0.0051792 secs]
```

图 5.32　实战案例 3 部分 GC 日志

通过 GC 日志可以看到，在本次实验中，GC 数量大幅度减少，并且存在的基本都是新生代 GC。从聚合报告中也可以看到，吞吐量为 674 次/秒，如图 5.33 所示。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	4	4	6	2	123	0.00%	674.7/sec	5151.2
总体	10000	4	4	6	2	123	0.00%	674.7/sec	5151.2

图 5.33　实战案例 3 聚合报告

5.7.7　实战案例 4——使用 ParrellOldGC 回收器

使用 ParrellOldGC 代替串行回收器。在笔者计算机上，默认情况下使用的是串行回收器。设置虚拟机参数如下：

```
set CATALINA_OPTS=-Xmx512m -Xms64m
-XX:MaxPermSize=32M
-Xloggc:gc.log -XX:+PrintGCDetails
-XX:+UseParallelGC -XX:+UseParallelOldGC -XX:ParallelGCThreads=4
```

堆大小和实战案例 3 保持不变，但是更换了垃圾回收器。测试得到的聚合报告如图 5.34 所示。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	4	4	6	2	124	0.00%	669.0/sec	5107.8
总体	10000	4	4	6	2	124	0.00%	669.0/sec	5107.8

图 5.34　实战案例 4 聚合报告

可以看到，由于 GC 压力在本次实验中并不大（由实验 3 的 GC 日志可以得出这个结论），因此更换垃圾回收器后，吞吐量并无改善。

5.7.8　实战案例 5——使用较小堆提高 GC 压力

通过设置一个较小的堆，增加 GC 压力，以此来考察不同垃圾回收器的表现，首先依然使用串行回收器。设置虚拟机参数如下：

```
set CATALINA_OPTS=-Xmx40m -Xms40m
-XX:MaxPermSize=32M
-Xloggc:gc.log -XX:+PrintGCDetails
```

将初始堆和最大堆统一设置为 40MB，并使用串行回收器。得到的聚合报告如图 5.35 所示。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	4	4	5	2	209	0.00%	646.0/sec	4932.6
总体	10000	4	4	5	2	209	0.00%	646.0/sec	4932.6

图 5.35　实战案例 5 聚合报告

减小堆大小后，吞吐量有较为明显的下降（从实战案例 3 可以知道，本次实验堆大小在 60MB 左右是没有 GC 压力的，而小于 60MB 的堆会有一定的 GC 压力）。

5.7.9　实战案例 6——测试 ParallelOldGC 的表现

在实战案例 5 的基础上，更换 ParallelOldGC 回收器进行测试。使用参数如下：

```
set CATALINA_OPTS=-Xmx40m -Xms40m
-XX:MaxPermSize=32M
-Xloggc:gc.log -XX:+PrintGCDetails
-XX:+UseParallelOldGC -XX:ParallelGCThreads=4
```

得到的聚合报告如图 5.36 所示。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	4	4	5	2	174	0.00%	685.2/sec	5231.7
总体	10000	4	4	5	2	174	0.00%	685.2/sec	5231.7

图 5.36　实战案例 6 聚合报告

可以看到，在有一定 GC 压力的情况下，使用 ParallelOldGC 回收器对系统性能有较为明显的改善。

注意：ParallelOldGC 回收器在多核 CPU 上才有可能改善性能，对于单核 CPU 或者并行能力较弱的计算机，还是应该选择串行回收器。

5.7.10　实战案例 7——测试 ParNew 回收器的表现

使用 ParNew 回收器代替 ParallelOldGC，这样在老年代依然使用串行回收器，仅在新生代使用并行。

```
set CATALINA_OPTS=-Xmx40m -Xms40m
-XX:MaxPermSize=32M
-Xloggc:gc.log -XX:+PrintGCDetails
-XX:+UseParNewGC
```

得到的聚合报告如图 5.37 所示。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	4	4	5	2	120	0.00%	660.0/sec	5039.0
总体	10000	4	4	5	2	120	0.00%	660.0/sec	5039.0

图 5.37　实战案例 7 聚合报告

可以看到，性能比实战案例 6 差些，但仍然好于实战案例 5 的全串行结果。

5.7.11　实战案例 8——测试 JDK 1.6 的表现

在本次实战案例中，使用 JDK 1.6，并使用如下参数：

```
  -Xmx40m -Xms40m -XX:MaxPermSize=32M  -Xloggc:gc.log -XX:+PrintGCDetails
-XX:+UseSerialGC
```

得到的聚合报告如图 5.38 所示。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	4	4	6	2	160	0.00%	650.9/sec	4972.7
总体	10000	4	4	6	2	160	0.00%	650.9/sec	4972.7

图 5.38　实战案例 8 聚合报告

5.7.12　实战案例——使用高版本虚拟机提升性能

在本次实战案例中，使用和实战案例 8 相同的 Java 虚拟机参数，但是使用 JDK 1.7 运行 Tomcat 服务器，得到如图 5.39 所示的结果。

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
HTTP请求	10000	4	4	6	2	157	0.00%	684.3/sec	5228.2
总体	10000	4	4	6	2	157	0.00%	684.3/sec	5228.2

图 5.39　实战案例 9 聚合报告

可以看到，只是简单的升级虚拟机版本，也可以得到意外的性能提升。这种免费的午餐是值得大家尝试的，但是 JDK 版本升级依然伴随有一定风险，也许在新版本的 JDK 中引入了某些未知的 Bug。因此，也必须要做好充分的回归测试工作。

5.8　小结

本章主要介绍了虚拟机支持的各种垃圾回收的种类、使用方法和相关参数，着重介绍了串行回收器、并行回收器、CMS 和 G1 回收器。此外，本章还介绍了有关对象分配和回收的一些细节问题。最后，笔者尝试使用各种垃圾回收器以及各项参数，测试了它们对 Tomcat 服务器的性能影响。

第 6 章 性能监控工具

性能是任何一款软件都需要关注的重要指标。除了软件的基本功能外，性能可以说是评价软件优劣的最重要指标之一。我们该如何有效地监控和诊断性能问题呢？本章基于实践，着重介绍了一些针对系统和 Java 虚拟机的监控和诊断工具，以帮助读者在实际开发过程中改善系统性能问题。

本章涉及的主要知识点有：

- Linux 下的性能监控工具。
- Windows 下的性能监控工具。
- JDK 自带的命令行工具。
- JConsole、Visual VM 和 Mission Control 的介绍。

6.1 有我更高效：Linux 下的性能监控工具

Linux 平台是使用最为广泛的服务器平台之一。不少 Java 端程序都运行在类 Linux 平台下（如 AIX、Solaris 等）。不同的类 Linux 操作系统间的很多命令都非常相似，不少命令仅有一些细节上的差异。本节主要介绍用于 Linux 平台下的性能收集和统计工具。

6.1.1 显示系统整体资源使用情况——top 命令

top 命令是 Linux 下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况。top 命令的部分输出如下：

```
[root@redhat6 tmp]# top
top - 09:33:57 up  6:37,  3 users,  load average: 0.10, 0.05, 0.07
Tasks: 191 total,   1 running, 190 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.3%us,  2.2%sy,  0.0%ni, 94.5%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2071628k total,  1738012k used,   333616k free,    75044k buffers
Swap:  4161528k total,     5348k used,  4156180k free,  1095040k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2078 root      20   0 56016  25m 8348 S  9.3  1.3   4:39.27 Xorg
 4022 oracle    20   0 82540  13m 9.9m S  1.3  0.7   0:11.28 gnome-terminal
16055 root      20   0  2660 1144  872 R  0.3  0.1   0:00.03 top
    1 root      20   0  2828 1272 1068 S  0.0  0.1   0:02.02 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.74 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.40 migration/1
    7 root      20   0     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/1
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/1
    9 root      20   0     0    0    0 S  0.0  0.0   0:00.13 events/0
```

top 命令的输出可以分为两个部分：前半部分是系统统计信息，后半部分是进程信息。

在统计信息中，第 1 行是任务队列信息，它的结果等同于 uptime 命令。从左到右依次表示：系统当前时间、系统运行时间、当前登录用户数。最后的 load average 表示系统的平均负载，即任务队列的平均长度，这 3 个值分别表示 1 分钟、5 分钟、15 分钟到现在的平均值。

第 2 行是进程统计信息，分别有正在运行的进程数、睡眠进程数、停止的进程数、僵尸进程数。

第 3 行是 CPU 统计信息，us 表示用户空间 CPU 占用率、sy 表示内核空间 CPU 占用率、ni 表示用户进程空间改变过优先级的进程 CPU 的占用率、id 表示空闲 CPU 占用率、wa 表示等待输入输出的 CPU 时间百分比、hi 表示硬件中断请求、si 表示软件中断请求。在 Mem 行中，从左到右，依次表示物理内存总量、已使用的物理内存、空闲物理内存、内核缓冲使用量。Swap 行依次表示交换区总量、空闲交换区大小、缓冲交换区大小。

top 命令的第 2 部分是进程信息区，显示了系统内各个进程的资源使用情况。在这张表格中，主要字段的含义如下。

- PID：进程 id。
- USER：进程所有者的用户名。
- PR：优先级。
- NI：nice 值，负值表示高优先级，正值表示低优先级。
- %CPU：上次更新到现在的 CPU 时间占用百分比。
- TIME+：进程使用的 CPU 时间总计，单位 1/100 秒。
- %MEM：进程使用的物理内存百分比。
- VIRT：进程使用的虚拟内存总量，单位 kb，VIRT=SWAP+RES。
- RES：进程使用的、未被换出的物理内存大小，单位 kb，RES=CODE+DATA。
- SHR：共享内存大小，单位 kb。
- COMMAND：命令名/命令行。

注意：使用 top 命令可以从宏观上观察系统各个进程对 CPU 的占用情况，以及内存使用情况。

6.1.2 监控内存和 CPU——vmstat 命令

vmstat 也是一款功能比较齐全的性能监测工具。它可以统计 CPU、内存使用情况、swap 使用情况等信息。和 sar 工具类似，vmstat 也可以指定采样周期和采样次数。

【示例 6-1】下例每秒采样一次，共计 3 次。

```
[root@redhat6 tmp]# vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0   5312 221444 119176 1122592   73    0     0    34  185  281  1  2 97  1  0
 0  0   5312 221048 119176 1122592   32    0     0     0  442  725  2  1 97  0  0
 0  0   5312 221048 119176 1122592    0    0     0     0  463  741  2  1 97  0  0
```

输出结果中，各个列含义如表 6.1 所示。

表 6.1　vmstat命令输出的含义

Procs	r：等待运行的进程数 b：处在非中断睡眠状态的进程数
Memory	swpd：虚拟内存使用情况，单位：KB free：空闲的内存，单位KB buff：被用来做为缓存的内存数，单位：KB
Swap	si：从磁盘交换到内存的交换页数量，单位：KB/秒 so：从内存交换到磁盘的交换页数量，单位：KB/秒
IO	bi：发送到块设备的块数，单位：块/秒 bo：从块设备接收到的块数，单位：块/秒
System	in：每秒的中断数，包括时钟中断 cs：每秒的上下文切换次数
CPU	us：用户CPU使用时间 sy：内核CPU系统使用时间 id：空闲时间

【示例 6-2】以下代码显示了一个线程切换频繁的 Java 程序。

```
public class HoldLockMain {
    public static Object[] lock=new Object[10];
    public static java.util.Random r=new java.util.Random();
    static{
        for(int i=0;i<lock.length;i++)
            lock[i]=new Object();
    }
    public static class HoldLockTask implements Runnable{    //一个持有锁的线程
        private int i;
        public HoldLockTask(int i){
            this.i=i;
        }
        @Override
        public void run() {
            try{
                while(true){
                    synchronized (lock[i]) {                //持有锁
                        if(i%2==0)
                            lock[i].wait(r.nextInt(10));     //等待
                        else
                            lock[i].notifyAll();            //通知
                    }
```

```
                }
            }catch(Exception e){
            }
        }
    }

    public static void main(String[] args){
        for(int i=0;i<lock.length*2;i++)              //每 2 个线程，使用同一锁对象
            new Thread(new HoldLockTask(i/2)).start();
    }
}
```

使用 vmstat 工具监控上述 Java 程序执行时的情况：

```
 [oracle@redhat6 ~]$ vmstat 1 4
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 6  0   5472 185240 134860 1257524   65    0     0    24  192  302  2  1 96  0  0
 7  0   5472 185108 134860 1257524    0    0     0     0 2399 2451 99  1  0  0  0
 9  0   5472 185124 134860 1257524    0    0     0     0 2350 2410 99  1  0  0  0
 7  0   5472 185124 134860 1257524  128    0     0     0 2339 2367 100 1  0  0  0
```

可以看到加粗部分有着很高的 cs 值（上下文切换）和 us 值（用户 CPU 时间），表明系统的上下文切换频繁，用户 CPU 占用率很高。

注意：vmstat 工具可以查看内存、交互分区、I/O 操作、上下文切换、时钟中断，以及 CPU 的使用情况。

6.1.3 监控 IO 使用——iostat 命令

iostat 可以提供详尽的 I/O 信息，它的基本使用如下：

```
[root@redhat6 tmp]# iostat 1 2
Linux 2.6.32-71.el6.i686 (redhat6)     03/25/2012     _i686_    (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.05    0.00    1.67    0.63    0.00   96.64

Device:            tps   kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               7.21       62.88        35.64    2083412    1180818
dm-0             13.29       62.66        35.41    2076369    1173160
dm-1              0.09        0.11         0.23       3720       7640
```

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.49    0.00   11.94    0.00    0.00   86.57

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               7.00         4.00        16.00          4         16
dm-0              8.00         4.00        16.00          4         16
dm-1              0.00         0.00         0.00          0          0
```

以上命令显示了 CPU 的使用概况和磁盘 I/O 的信息。输出信息每 1 秒采样 1 次，合计采样两次。如果只需要显示磁盘情况，不需要显示 CPU 使用情况，则可以使用命令：

```
iostat -d 1 2
```

-d 表示输出磁盘使用情况。输出结果中各个列的含义如下。

- tps：该设备每秒的传输次数。
- kB_read/s：每秒从设备读取的数据量。
- kB_wrtn/s：每秒向设备写入的数据量。
- kB_read：读取的总数据量。
- kB_wrtn：写入的总数据量。

如果需要得到更多的统计信息，可以使用-x 选项，如：

```
iostat -x 1 2
```

注意：磁盘 I/O 很容易成为系统性能瓶颈，通过 iostat 可以快速定位系统是否产生了大量的 I/O 操作。

6.1.4　多功能诊断器——pidstat 工具

pidstat 是一个功能强大的性能监测工具，它也是 Sysstat 的组件之一。读者可以在 http://www.icewalkers.com/Linux/Software/59040/Sysstat.html 下载这个工具。下载后，通过./configure、make、make install 等 3 个命令即可安装 pidstat 工具。如果是 Ubuntu 的系统，也可以简单地通过以下命令安装：

```
sudo apt-get install sysstat
```

命令行工具 pidstat 的强大之处在于，它不仅可以监视进程的性能情况，也可以监视线程的性能情况。本节将详细介绍 pidstat 这方面的功能。

1. CPU 使用率监控

【示例 6-3】下例是一个简单地占用 CPU 的程序，它开启了 4 个用户线程，其中，1 个线

程大量占用 CPU 资源，其他 3 个线程则处于空闲状态。

```
public class HoldCPUMain {
    public static class HoldCPUTask implements Runnable{
        @Override
        public void run() {
            while(true){
                double a=Math.random()*Math.random();     //占用 CPU
            }
        }
    }
    public static class LazyTask implements Runnable{
        public void run(){
            try{
                while(true){
                    Thread.sleep(1000);                   //空闲线程
                }
            }catch(Exception e){

            }
        }
    }

    public static void main(String[] args){
        new Thread(new HoldCPUTask()).start();       //开启线程，占用 CPU
        new Thread(new LazyTask()).start();          //空闲线程
        new Thread(new LazyTask()).start();
        new Thread(new LazyTask()).start();
    }
}
```

运行以上程序，要监控该程序的 CPU 使用率，可以先使用 jps 命令找到 Java 程序的 PID，然后使用 pidstat 命令输出程序的 CPU 使用情况。

```
[root@redhat6 tmp]# jps
443 Jps
16185
1187 HoldCPUMain
[root@redhat6 tmp]# pidstat -p 1187 -u 1 3
Linux 2.6.32-71.el6.i686 (redhat6)    03/25/2012    _i686_    (2 CPU)

01:38:19 PM       PID    %usr %system  %guest    %CPU   CPU  Command
01:38:20 PM      1187   99.01    0.00    0.00   99.01     1  java
01:38:21 PM      1187  100.00    0.00    0.00  100.00     1  java
```

```
01:38:22 PM       1187   99.00    0.00    0.00   99.00     1  java
Average:           1187   99.34    0.00    0.00   99.34     -  java
```

pidstat 的参数-p 用于指定进程 ID，-u 表示对 CPU 使用率的监控。参数 1 3 表示每秒钟采样一次，合计采样 3 次。从这个输出中可以看到，该应用程序 CPU 占用率几乎达 100%。pidstat 的功能不仅仅限于观察进程信息，它可以进一步监控线程的信息。使用以下命令：

```
pidstat -p 1187 1 3 -u -t
```

这个命令的部分输出如下：

```
01:47:30 PM      TGID       TID    %usr %system  %guest    %CPU   CPU  Command
01:47:31 PM      1187         -   98.02    0.00    0.00   98.02     0  java
01:47:31 PM         -      1187    0.00    0.00    0.00    0.00     0  |__java
省略部分线程
01:47:31 PM         -      1203    0.00    0.00    0.00    0.00     0  |__java
01:47:31 PM         -      1204   97.03    0.00    0.00   97.03     1  |__java
省略部分线程
01:47:31 PM         -      1207    0.00    0.00    0.00    0.00     1  |__java
```

-t 参数将系统性能的监控细化到线程级别。从这个输出中可以知道，该 Java 应用程序之所以占用如此之高的 CPU，是因为线程 1204 的缘故。

注意：使用 pidstat 工具不仅可以定位到进程，甚至可以进一步定位到线程。

使用以下命令可以导出指定 Java 应用程序的所有线程：

```
jstack -l 1187 >/tmp/t.txt
```

在输出的 t.txt 文件中，可以找到这么一段输出内容：

```
"Thread-0" prio=10 tid=0xb75b3000 nid=0x4b4 runnable [0x8f171000]
  java.lang.Thread.State: RUNNABLE
   at javatuning.ch6.toolscheck.HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:7)
   at java.lang.Thread.run(Thread.java:636)

  Locked ownable synchronizers:
   - None
```

从加粗的文字中可以看到，这个线程正是 HoldCPUTask 类，它的 nid（native ID）为 0x4b4，转为 10 进制后，正好是 1204。

通过这个方法，开发人员可以使用 pidstat 很容易地捕获到在 Java 应用程序中大量占用 CPU 的线程。

2. I/O 使用监控

磁盘 I/O 也是常见的性能瓶颈之一，使用 pidstat 也可以监控进程内线程的 I/O 情况。

【示例 6-4】下例开启了 4 个线程，其中线程 HoldIOTask 产生了大量的 I/O 操作。

```
public class HoldIOMain {
    public static class HoldIOTask implements Runnable{
        @Override
        public void run() {
            while(true){
                try {
                    FileOutputStream fos=new FileOutputStream(new File("temp"));
                    for(int i=0;i<10000;i++)
                        fos.write(i);                    //大量的写操作
                    fos.close();
                    FileInputStream fis=new FileInputStream(new File("temp"));
                    while(fis.read()!=-1);               //大量的读操作
                } catch (FileNotFoundException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    public static class LazyTask implements Runnable{
        public void run(){
            try{
                while(true){
                    Thread.sleep(1000);                  //一个空闲线程
                }
            }catch(Exception e){

            }
        }
    }

    public static void main(String[] args){
        new Thread(new HoldIOTask()).start();            //开启占用 I/O 的线程
        new Thread(new LazyTask()).start();              //开启空闲线程
        new Thread(new LazyTask()).start();
        new Thread(new LazyTask()).start();
    }
}
```

在程序运行过程中，使用以下命令监控程序 I/O 使用情况。其中 22796 是通过 jps 命令查询到的进程 ID，-d 参数表明监控对象为磁盘 I/O。1 3 表示每秒钟采样一次，合计采样 3 次。

```
[oracle@redhat6 ~]$ pidstat -p 22796 -d -t 1 3
Linux 2.6.32-71.el6.i686 (redhat6)    03/25/2012    _i686_    (2 CPU)

06:06:00 PM     TGID      TID  kB_rd/s   kB_wr/s kB_ccwr/s  Command
06:06:01 PM    22796        -     0.00    332.00      0.00  java
06:06:01 PM        -    22796     0.00      0.00      0.00  |__java
06:06:01 PM        -    22799     0.00      0.00      0.00  |__java
06:06:01 PM        -    22802     0.00      0.00      0.00  |__java
06:06:01 PM        -    22803     0.00      0.00      0.00  |__java
06:06:01 PM        -    22805     0.00      0.00      0.00  |__java
06:06:01 PM        -    22806     0.00      0.00      0.00  |__java
06:06:01 PM        -    22807     0.00      0.00      0.00  |__java
06:06:01 PM        -    22808     0.00      0.00      0.00  |__java
06:06:01 PM        -    22809     0.00      0.00      0.00  |__java
06:06:01 PM        -    22810     0.00      0.00      0.00  |__java
06:06:01 PM        -    22811     0.00      0.00      0.00  |__java
06:06:01 PM        -    22812     0.00      0.00      0.00  |__java
06:06:01 PM        -    22813     0.00    328.00      0.00  |__java
06:06:01 PM        -    22814     0.00      0.00      0.00  |__java
06:06:01 PM        -    22815     0.00      0.00      0.00  |__java
06:06:01 PM        -    22816     0.00      0.00      0.00  |__java
```

从输出结果中可以看到，进程中的 22813（0x591D）线程产生了大量 I/O 操作。通过前文中提到的 jstatck 命令，可以导出当前线程堆栈，查找 nid 为 22813（0x591D）的线程，即可定位到 HoldIOTask 线程。

注意：使用 pidstat 命令可以查看进程和线程的 I/O 信息。

3. 内存监控

使用 pidstat 命令，还可以监控指定进程的内存使用情况。

【示例 6-5】下例使用 pidstat 工具对进程 ID 为 27233 的进程进行内存监控。每秒钟刷新一次，共进行 5 次统计。

```
[oracle@redhat6 ~]$ pidstat -r -p 27233 1 5
Linux 2.6.32-71.el6.i686 (redhat6)    04/14/2012    _i686_    (2 CPU)

09:50:32 AM       PID  minflt/s  majflt/s     VSZ    RSS   %MEM  Command
```

```
09:50:33 AM     27233     0.00       0.00  728164  11476   0.55  java
09:50:34 AM     27233     1.00       0.00  728164  11480   0.55  java
09:50:35 AM     27233     1.00       0.00  728164  11484   0.55  java
09:50:36 AM     27233     1.00       0.00  728164  11488   0.55  java
09:50:37 AM     27233     0.99       0.00  728164  11492   0.55  java
Average:        27233     0.80       0.00  728164  11484   0.55  java
```

输出结果中各列含义如下。

- minflt/s：表示该进程每秒 minor faults (不需要从磁盘中调出内存页）的总数。
- majflt/s：表示该进程每秒 major faults（需要从磁盘中调出内存页）的总数。
- VSZ：表示该进程使用的虚拟内存大小，单位为 KB。
- RSS：表示该进程占用的物理内存大小，单位为 KB。
- %MEM：表示占用内存比率。

注意：pidstat 工具是一款多合一的优秀工具。它不仅可以监控 CPU、I/O 和内存资源，甚至可以将问题定位到相关线程，方便应用程序的故障排查。

6.2 用我更高效：Windows 下的性能监控工具

作为桌面市场的占领者，在 Windows 平台上也运行着大量的 Java 应用程序。本节主要介绍一些可以工作在 Windows 平台上的性能监控工具，包括 Windows 系统自带的任务管理、性能监控工具，以及一些优秀的第三方工具。本节以 Windows 7 为例，介绍一些在 Windows 下的性能监控工具。

6.2.1 任务管理器

Windows 系统的任务管理器是大家最为熟知的一款系统工具。通过 Ctrl+Alt+Del 组合键便能呼出。任务管理乍看之下并不起眼，但事实上它却是使用最为方便，功能也非常强大的一款性能统计工具。任务管理的界面如图 6.1 所示。

可以看到，任务管理的进程页罗列了系统内进程的名称、所属用户、CPU 使用率和内存使用量等信息。通过任务管理器，可以方便地实时监控各个进程的 CPU 和内存使用情况，对于简单的性能监控，已经足够了。

除了图 6.1 中所展示的列外，任务管理的进程页面还可以显示更多的性能参数，比如进程 ID、I/O 信息等，如图 6.2 所示。通过“查看”菜单下的“选择列”菜单，可以弹出“选择进程

页列”对话框。在这个对话框中，可以选择需要监控的列信息。



图 6.1　任务管理器界面

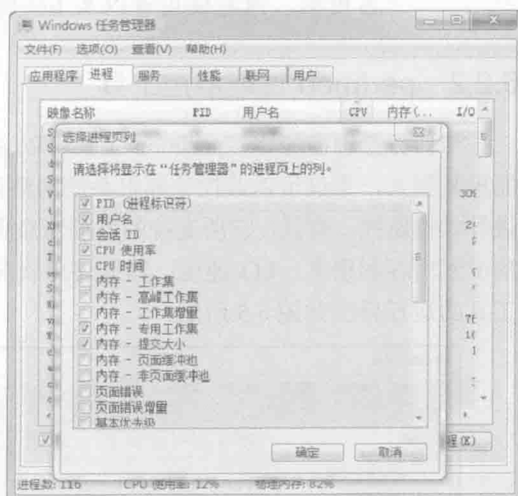


图 6.2　任务管理器支持的统计类型

除了对进程进行单独的监控外，任务管理还能对计算机系统的整体运行情况进行监控，包括 CPU、内存以及网络的使用情况。图 6.3 展示了 CPU 的实际使用率和内存使用情况，图 6.4 展示了当前的网络使用情况。

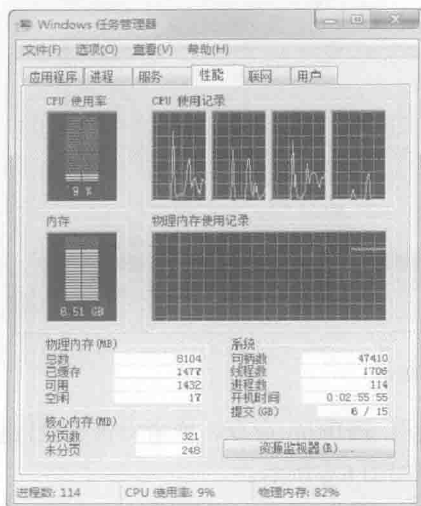


图 6.3　CPU 使用情况



图 6.4　网络使用情况

注意：任务管理器很常用，也非常强大。它可以显示系统的网络负载、任意进程的 CPU 占用率、内存使用量以及 I/O 使用情况。

6.2.2 perfmon 性能监控工具

与任务管理器相比，perfmon 工具可以说是 Windows 的专业级的性能监控工具了。它的功能极其强大，不仅可以监控计算机系统的整体运行情况，也可以专门针对某一个进程或者线程进行状态监控。并且，它所支持的监控对象也非常多，几乎涉及系统的所有方面，从 CPU 使用情况到内存利用率、I/O 速度、网络使用状况、进程数、线程等无不在其监控范围内。perfmon 工具的运行界面如图 6.5 所示。

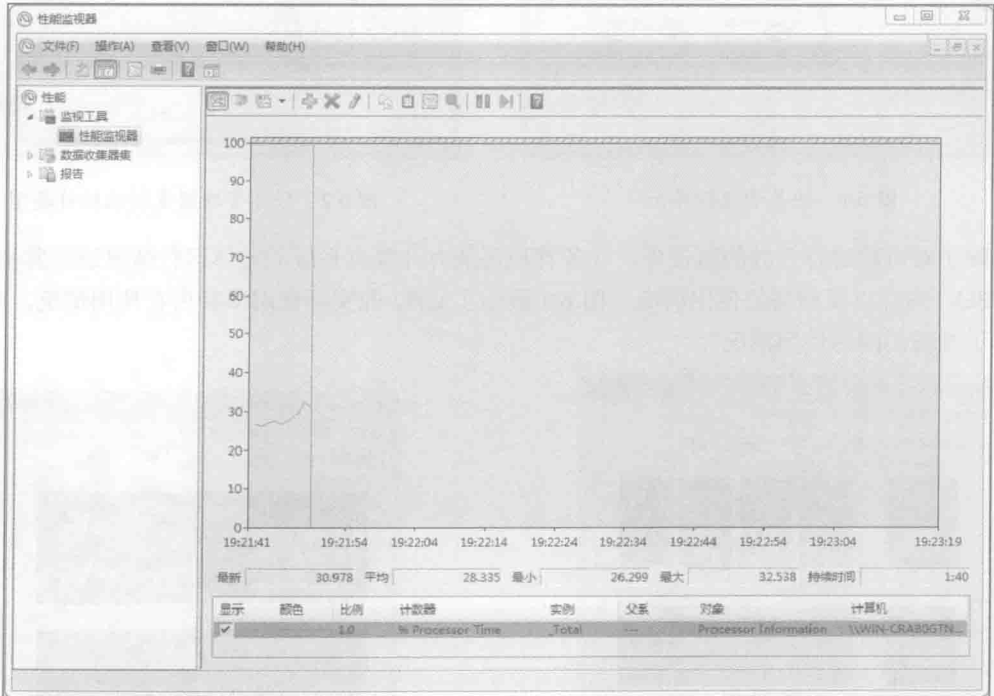


图 6.5 perfmon 运行界面

读者可以在任务栏开始菜单的“运行”对话框中使用 perfmon 命令，或者双击控制面板的管理工具中的“性能监视器”快捷方式打开这个工具，如图 6.6 所示。

perfmon 工具的底部表格显示了当前正在监控的系统对象。可以在监控对象表格中使用右键添加计数器，如图 6.7 所示，选择需要监控的对象，如图 6.8 所示。



图 6.6　打开 perfmon 的方式



图 6.7　perfmon 中添加计数器



图 6.8　选择要监控的指标和监控对象

可以看到，perfmon 工具支持的性能对象很多。这里，需要重点关注的是 Thread 监控项下的内容，如 User Time 表示线程占用 CPU 的时间百分比、ID Thread 表示线程的 ID 号、ID Process 表示进程 ID 号。在监控的对象实例中，可以选择 Java 进程的线程。

【示例 6-6】读者应该还记得在介绍 6.1.4 节中使用的 HoldCPUMain 程序吧！以该样例程序为例，本节将展示如何使用 perfmon 工具找出在 Java 应用程序中最消耗 CPU 资源的线程。

（1）运行 HoldCPUMain。然后按照图 6.8 的配置，设置测量对象为 Thread，并选择 javaw.exe 进程中所有的线程。

（2）单击查看报表按钮，切换 perfmon 工具的显示模式，如图 6.9 所示。

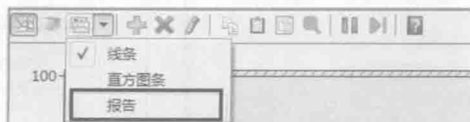


图 6.9　切换显示模式

（3）在报表中可以很容易地找到，在当前 Java 应用程序中，线程 ID 为 2548 的线程占用了很高的 CPU，如图 6.10 所示。

Thread	javaw 0	javaw 1	javaw 10	javaw 11	javaw 12
% User Time	0.000	0.000	98.663	0.000	0.000
ID Thread	5,200.000	11,324.000	11,452.000	10,468.000	4,504.000

图 6.10　线程监控结果

（4）将 11452 换算成 16 进制为 2CBC。通过 jstack 等工具导出 Java 应用程序的线程快照，并查找 2CBC，可以找到：

```
"Thread-0" prio=6 tid=0x04d98400 nid=0x2cbc runnable [0x04d2f000]
   java.lang.Thread.State: RUNNABLE
    at geym.zbase.ch6.hold.HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:8)
    at java.lang.Thread.run(Thread.java:724)
```

至此完成线程定位，通过 perfmon 找到了 Java 程序中消耗 CPU 最多的线程代码。

注意： perform 工具也是 Windows 自带的一款性能监控软件。可以监控的性能指标繁多，功能也非常强大。当任务管理无法满足要求时，推荐使用。

在 Windows 7 下，perfmon 工具支持带参数/res 启动，用于专门监控系统资源的使用情况。使用如下命令启动 perfmon：

```
perfmon /res
```

使用该命令打开的资源监控工具可以非常细致地观察系统内的资源使用情况，如图 6.11 所示。在这个视图中，可以查看每个 CPU 核的信息、磁盘使用率、网络使用率、内存使用率、系统中打开和监听的端口及相关进程等。



图 6.11　perfmon /res 的界面

6.2.3　Process Explorer 进程管理工具

Process Explorer 是一款功能极其强大的进程管理工具。它完全可以替代 Windows 自带的任务管理器。读者可以在 http://technet.microsoft.com/en-us/sysinternals/bb896653 下载该工具。Process Explorer 的运行主界面如图 6.12 所示。

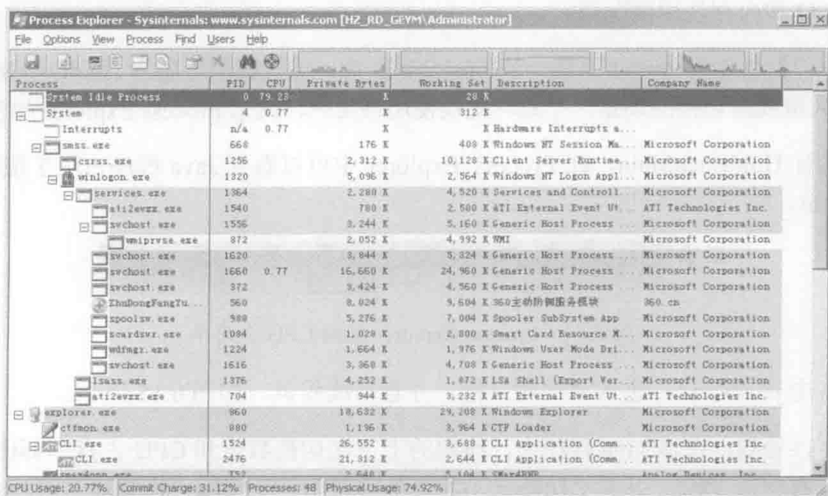


图 6.12　Process Explorer 运行界面

Process Explorer 不仅显示了系统内所有的进程，还进一步显示了进程间的父子关系。Process Explorer 所支持的测量项非常多。可以在表格头部单击右键，打开“选择列”对话框选择需要统计的策略项，如图 6.13 所示。Process Explorer 将统计项进行了分类，主要有进程模块信息、进程性能、I/O 使用情况、网络使用情况、内存使用情况等选项卡。

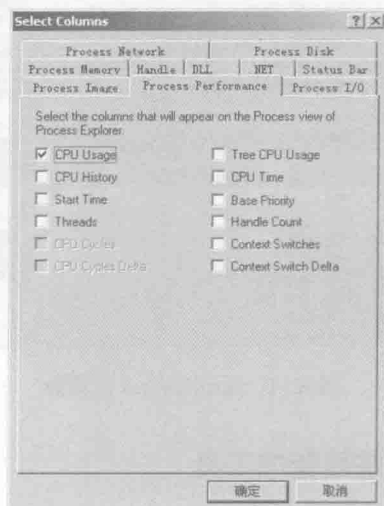


图 6.13 进程性能统计类型

在进程中右击，选择属性对话框，可以显示选中进程的详细信息。包括线程数、上下文环境、CPU、I/O 和内存使用情况等。

【示例 6-7】在介绍 vmstat 工具一节（6.1.2 节）中，读者使用了名为 HoldLockMain 的例子，演示了一个大量线程切换的场景。在此，继续使用该实例，演示 Process Explorer 的使用方法。

（1）运行 HoldLockMain，在 Process Explorer 中可以看到 Java 程序占用了很高的 CPU，如图 6.14 所示。

eclipse.exe	300		141,372 K
javaw.exe	3296	99.18	28,772 K

图 6.14 Process Explorer 监控 CPU 占用率

（2）右击 javaw.exe，进入属性对话框，并查看线程页，如图 6.15 所示。

（3）可以看到，在该 Java 程序中，线程的上下文切换数值和 CPU 占有率都比较大。查看当前 CPU 占有率达 12%，上下文切换达 163 次的线程 ID 为 2864，换算成 16 进制为 B30。在线程快照中查找，有以下结果：

```
"Thread-14" prio=6 tid=0x02b4c800 nid=0xb30 runnable [0x032bf000..0x032bfc94]
   java.lang.Thread.State: RUNNABLE
    at javatuning.ch6.toolscheck.HoldLockMain$HoldLockTask.run
(HoldLockMain.java:19)
    - locked <0x22a23478> (a java.lang.Object)
    at java.lang.Thread.run(Thread.java:619)

   Locked ownable synchronizers:
    - None
```

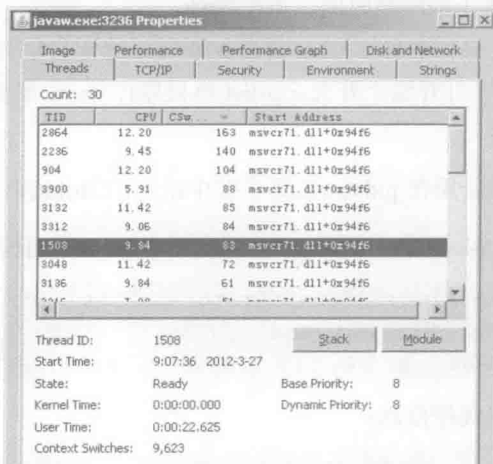


图 6.15　查看线程情况

通过 Process Explorer，这里已经找到其中一个消耗 CPU 资源的线程。在实际开发过程中，使用类似的方法，便可以找出程序中最为消耗资源的线程。

注意：Process Explorer 可以认为是一个加强版的任务管理器，但是它不是 Windows 系统自带的工具，需要下载安装。

6.2.4　pslist 命令——Windows 下也有命令行工具

与之前介绍的性能监控工具不同，pslist 工具是一款 Windows 下的命令行工具。虽然与之前的 GUI 工具相比，命令行工具在易用性上略有不足，但它依然具有 GUI 工具所不能替代的功能与用途。读者可以在 http://technet.microsoft.com/en-us/sysinternals/bb896682 下载并安装该工具。

pslist 的基本用法如下：

```
pslist [-d][-m][-x][-t][-s [n] [-r n] [name|pid]
```

各参数含义如下。

- -d：显示线程详细信息。
- -m：显示内存详细信息。
- -x：显示进程、内存和线程信息。
- -t：显示进程间父子关系。
- -s [n]：进入监控模式，n 指定程序运行时间，使用 Esc 键退出。
- -r n：指定监控模式下的刷新时间，单位为秒。
- name：指定监控的进程名称，pslist 将监控所有以给定名字开头的进程。
- -e：使用精确匹配，打开这个开关，pslist 将只监控 name 参数指定的进程。
- pid：指定进程 ID。

【示例 6-8】本节依然延用在 pidstat 工具章节中给出的 HoldCPUTask 示例。

（1）运行 HoldCPUTask 程序，使用 pslist 列出所有 Java 应用程序进程：

```
C:\Documents and Settings\Administrator>pslist java
Name              Pid  Pri Thd  Hnd   Priv        CPU Time    Elapsed Time
javaw            2268   8  14  119  28068      0:02:08.765     0:02:10.265
```

（2）使用-d 参数列出线程信息：

```
C:\Documents and Settings\Administrator>pslist java -d

javaw 2268:
 Tid Pri    Cswtch          State     User Time   Kernel Time   Elapsed Time
3212   9        49     Wait:UserReq  0:00:00.015  0:00:00.000    0:03:34.421
1428   9       280     Wait:UserReq  0:00:00.078  0:00:00.078    0:03:34.375
2052  10       274     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:34.171
 540  11         3     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:33.953
 184   9         2     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:33.953
4052  10         6     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:33.484
1156  10         2     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:33.484
2448  11        42     Wait:UserReq  0:00:00.015  0:00:00.000    0:03:33.484
3728   8         3     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:33.484
2552  10      4699   Wait:DelayExec  0:00:00.000  0:00:00.000    0:03:33.484
2548   8    142357          Running  0:03:32.406  0:00:00.015    0:03:33.328
1856   8       252     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:33.312
3528   8       250     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:33.312
1448   8       238     Wait:UserReq  0:00:00.000  0:00:00.000    0:03:33.312
```

（3）找到正在运行中并且具有最高 Cswtch 上下文切换值的线程 2548，换算成 16 进制为 9F4（很明显，该线程也是占用 CPU 时间最多的线程）。在线程快照中查找这个线程 ID：

```
"Thread-0" prio=6 tid=0x02b38800 nid=0x9f4 runnable [0x02e5f000..0x02e5fb94]
   java.lang.Thread.State: RUNNABLE
    at java.util.Random.nextDouble(Random.java:394)
    at java.lang.Math.random(Math.java:695)
    at javatuning.ch6.toolscheck.HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:8)
    at java.lang.Thread.run(Thread.java:619)

   Locked ownable synchronizers:
    - None
```

进行以上操作后，即可找到当前最消耗 CPU 的线程 HoldCPUTask。

6.3 外科手术刀：JDK 性能监控工具

在 JDK 的开发包中，除了大家熟知的 java.exe 和 javac.exe 外，还有一系列辅助工具。这些辅助工具可以帮助开发人员很好地解决 Java 应用程序的一些疑难杂症。这些工具在 JDK 安装目录下的 bin 目录中，图 6.16 显示了部分辅助工具。

虽然乍看之下，这些工作都是 exe 的可执行文件。但事实上，它们只是 Java 程序的一层包装，其真正实现是在 tools.jar 中，如图 6.17 所示。

jdb.exe	26 KB	应用程序
jhat.exe	26 KB	应用程序
jinfo.exe	26 KB	应用程序
jmap.exe	26 KB	应用程序
jps.exe	26 KB	应用程序
jrunscript.exe	26 KB	应用程序
jstack.exe	26 KB	应用程序
jstat.exe	26 KB	应用程序
jstatd.exe	26 KB	应用程序

图 6.16 JDK 内置工具

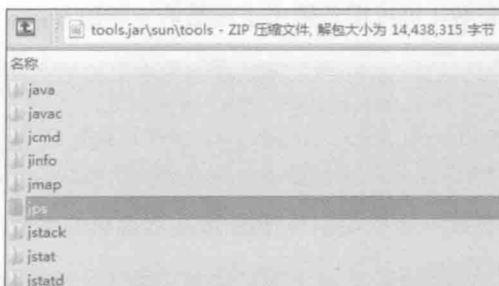


图 6.17 tools.jar 中的实现

以 jps 工具为例，在控制台执行 jps 命令和 java -classpath %Java_HOME%/lib/tools.jar sun.tools.jps.Jps 命令是等价的，即 jps.exe 只是这个命令的一层包装。

6.3.1 查看 Java 进程——jps 命令

命令 jps 类似于 linux 下的 ps，但它只用于列出 Java 的进程。直接运行 jps 不加任何参数，可以列出 Java 程序进程 ID 以及 Main 函数短名称，如下所示：

```
C:\Documents and Settings\Administrator>jps
6260 Jps
7988 Main
400
```

从这个输出中可以看到，当前系统中共存在 3 个 Java 应用程序，其中第一个输出 Jps 就是 jps 命令本身，这更加证明此命令的本质也是一个 Java 程序。此外，jps 还提供了一系列参数来控制它的输出内容。

参数-q 可以指定 jps 只输出进程 ID，而不输出类的短名称：

```
C:\Documents and Settings\Administrator>jps -q
7988
7152
```

参数-m 可以用于输出传递给 Java 进程（主函数）的参数：

```
C:\Documents and Settings\Administrator>jps -m
7988 Main --log-config-file D:\tools\squirrel-sql-3.2.1\log4j.properties --squir
rel-home D:\tools\squirrel-sql-3.2.1
7456 Jps -m
```

参数-l 可以用于输出主函数的完整路径：

```
C:\Documents and Settings\Administrator>jps -m -l
7244 sun.tools.jps.Jps -m -l
7988 net.sourceforge.squirrel_sql.client.Main --log-config-file D:\tools\squirre
l-sql-3.2.1\log4j.properties --squirrel-home D:\tools\squirrel-sql-3.2.1
```

参数-v 可以显示传递给 Java 虚拟机的参数：

```
C:\Documents and Settings\Administrator>jps -m -l -v
6992 sun.tools.jps.Jps -m -l -v -Denv.class.path=.;D:\tools\jdk6.0\lib\dt.jar;D:
\tools\jdk6.0\lib\tools.jar;D:\tools\jdk6.0\lib -Dapplication.home=D:\tools\jdk6
.0 -Xms8m
7988 net.sourceforge.squirrel_sql.client.Main --log-config-file D:\tools\squirre
l-sql-3.2.1\log4j.properties --squirrel-home D:\tools\squirrel-sql-3.2.1 -Xmx256
m -Dsun.java2d.noddraw=true
```

注意：jps 命令类似于 ps 命令，但是它只列出系统中所有的 Java 应用程序。通过 jps 命令可以方便地查看 Java 进程的启动类、传入参数和 Java 虚拟机参数等信息。

6.3.2 查看虚拟机运行时信息——jstat 命令

jstat 是一个可以用于观察 Java 应用程序运行时相关信息的工具。它的功能非常强大，可以通过它查看堆信息的详细情况。它的基本使用语法为：

```
jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]
```

- 选项 option 可以由以下值构成。
  - -class：显示 ClassLoader 的相关信息。
  - -compiler：显示 JIT 编译的相关信息。
  - -gc：显示与 GC 相关的堆信息。
  - -gccapacity：显示各个代的容量及使用情况。
  - -gccause：显示垃圾收集相关信息（同-gcutil），同时显示最后一次或当前正在发生的垃圾收集的诱发原因。
  - -gcnew：显示新生代信息。
  - -gcnewcapacity：显示新生代大小与使用情况。
  - -gcold：显示老年代和永久代的信息。
  - -gcoldcapacity：显示老年代的大小。
  - -gcpermcapacity：显示永久代的大小。
  - -gcutil：显示垃圾收集信息。
  - -printcompilation：输出 JIT 编译的方法信息。
- -t 参数可以在输出信息前加上一个 Timestamp 列，显示程序的运行时间。
- -h 参数可以在周期性数据输出时，输出多少行数据后，跟着输出一个表头信息。
- interval 参数用于指定输出统计数据的周期，单位为毫秒。
- count 用于指定一共输出多少次数据。

如下示例输出 Java 进程 2972 的 ClassLoader 相关信息。每秒钟统计一次信息，一共输出两次。

```
C:\Documents and Settings\Administrator>jstat -class -t 2972 1000 2
Timestamp       Loaded  Bytes  Unloaded  Bytes     Time
       1395.6    2375  2683.8         7    6.2     3.45
       1396.6    2375  2683.8         7    6.2     3.45
```

在-class 的输出中，Loaded 表示载入了类的数量，Bytes 表示载入类的合计大小，Unloaded 表示卸载类的数量，第 2 个 Bytes 表示卸载类的大小，Time 表示在加载和卸载类上所花的时间。

下例显示了查看 JIT 编译的信息：

```
C:\Documents and Settings\Administrator>jstat -compiler -t 2972
Timestamp       Compiled Failed Invalid  Time   FailedType FailedMethod
         1675.9      779      0       0  0.61          0
```

Compiled 表示编译任务执行的次数，Failed 表示编译失败的次数，Invalid 表示编译不可用的次数，Time 表示编译的总耗时，FailedType 表示最后一次编译失败的类型，FailedMethod 表示最后一次编译失败的类名和方法名。

下例显示了与 GC 相关的堆信息的输出：

```
C:\Documents and Settings\Administrator>jstat -gc 2972
 S0C     S1C    S0U    S1U   EC       EU       OC         OU         PC
PU      YGC    YGCT   FGC   FGCT    GCT
 64.0    64.0   0.0    2.0   896.0   448.9    12312.0   9019.1    12288.0  9101
.3   101        0.153  2     0.210   0.364
```

- S0C：s0（from）的大小（KB）。
- S1C：s1（from）的大小（KB）。
- S0U：s0（from）已使用的空间（KB）。
- S1U：s1（from）已使用的空间（KB）。
- EC：eden 区的大小（KB）。
- EU：eden 区的使用空间（KB）。
- OC：老年代大小（KB）。
- OU：老年代已经使用的空间（KB）。
- PC：永久区大小（KB）。
- PU：永久区使用空间（KB）。
- YGC：新生代 GC 次数。
- YGCT：新生代 GC 耗时。
- FGC：Full GC 次数。
- FGCT：Full GC 耗时。
- GCT：GC 总耗时。

下例显示了各个代的信息，与-gc 相比，它不仅输出了各个代的当前大小，也包含了各个代的最大值和最小值。

```
C:\Documents and Settings\Administrator>jstat -gccapacity 2972
 NGCMN    NGCMX     NGC     S0C   S1C       EC      OGCMN      OGCMX       OGC
OC       PGCMN    PGCMX     PGC       PC     YGC    FGC
  1024.0  20160.0   1024.0   64.0   64.0    896.0     4096.0   241984.0
12312
.0   12312.0  12288.0  65536.0  12288.0  12288.0    129    2
```

- NGCMN：新生代最小值（KB）。
- NGCMX：新生代最大值（KB）。
- NGC：当前新生代大小（KB）。
- OGCMN：老年代最小值（KB）。
- OGCMX：老年代最大值（KB）。
- PGCMN：永久代最小值（KB）。
- PGCMX：永久代最大值（KB）。

下例显示了最近一次 GC 的原因，以及当前 GC 的原因：

```
C:\Documents and Settings\Administrator>jstat -gccause 2972
 S0     S1     E      O      P     YGC    YGCT    FGC  FGCT    GCT       LGCC           GCC
 0.00 0.00  19.58  59.99  91.43  143    0.207  3    0.331  0.538   System.gc()   No GC
```

- LGCC：上次 GC 的原因。
- GCC：当前 GC 的原因。

本例显示，最近一次 GC 是由于显式的 System.gc()调用所引起的，当前时刻未进行 GC。

-gcnew 参数可以用于查看新生代的一些详细信息：

```
C:\Documents and Settings\Administrator>jstat -gcnew 2972
 S0C     S1C      S0U   S1U   TT   MTT  DSS      EC       EU      YGC    YGCT
 128.0  128.0    0.0   11.8  15   15   64.0  1024.0    139.8    159   0.223
```

- TT：新生代对象晋升到老年代对象的年龄。
- MTT：新生代对象晋升到老年代对象的年龄最大值。
- DSS：所需的 survivor 区大小。

-gcnewcapacity 参数可以详细输出新生代各个区的大小信息：

```
C:\Documents and Settings\Administrator>jstat -gcnewcapacity 2972
NGCMN  NGCMX     NGC     S0CMX   S0C     S1CMX   S1C     ECMX      EC       YGC    FGC
1024.0 20160.0  1280.0  128.0  1984.0  1984.0  128.0  16192.0  1024.0  178    3
```

- S0CMX：s0 区的最大值（KB）。
- S1CMX：s1 区的最大值（KB）。

- ECMX：eden 区的最大值（KB）。

-gcold 用于展现老年代 GC 的概况：

```
C:\Documents and Settings\Administrator>jstat -gcold 2972
 PC          PU          OC          OU         YGC     FGC    FGCT      GCT
 12288.0  11295.6    15048.0      9106.1     190      3    0.331    0.580
```

-gcoldcapacity 用于展现老年代的容量信息：

```
C:\Documents and Settings\Administrator>jstat -gcoldcapacity 2972
OGCMN       OGCMX          OGC           OC        YGC    FGC     FGCT      GCT
4096.0    241984.0     15048.0     15048.0    195      3    0.331    0.584
```

-gcpermcapacity 用于展示永久区的使用情况：

```
C:\Documents and Settings\Administrator>jstat -gcpermcapacity 2972
   PGCMN        PGCMX        PGC          PC        YGC    FGC     FGCT      GCT
   12288.0    65536.0    12288.0    12288.0    220      3    0.331    0.605
```

-gcutil 用于展示 GC 回收相关信息：

```
C:\Documents and Settings\Administrator>jstat -gcutil 2972
  S0      S1      E       O       P       YGC     YGCT    FGC     FGCT      GCT
  7.65   0.00  62.88  60.60  92.19    224    0.277     3    0.331    0.609
```

- S0：s0 区使用的百分比。
- S1：s1 区使用的百分比。
- E：eden 区使用的百分比。
- O：Old 区使用的百分比。
- P：永久区使用的百分比。

注意：jstat 命令可以非常详细地查看 Java 应用程序的堆使用情况以及 GC 情况。

6.3.3 查看虚拟机参数——jinfo 命令

jinfo 可以用来查看正在运行的 Java 应用程序的扩展参数，甚至支持在运行时，修改部分参数。它的基本语法为：

```
jinfo <option> <pid>
```

其中 option 可以为以下信息。

- -flag <name>：打印指定 Java 虚拟机的参数值。
- -flag [+|-]<name>：设置指定 Java 虚拟机参数的布尔值。

- -flag <name>=<value>：设置指定 Java 虚拟机参数的值。

在很多情况下，Java 应用程序不会指定所有的 Java 虚拟机参数。而此时，开发人员可能不知道某一个具体的 Java 虚拟机参数的默认值。在这种情况下，可能需要通过查找文档获取某个参数的默认值。这个查找过程可能是非常艰难的。但有了 jinfo 工具，开发人员可以很方便地找到 Java 虚拟机参数的当前值。

比如，下例显示了新生代对象晋升到老年代对象的最大年龄。在应用程序启动时，并没有指定这个参数，但通过 jinfo，可以查看这个参数的当前数值。

```
C:\Documents and Settings\Administrator>jinfo -flag MaxTenuringThreshold 2972
-XX:MaxTenuringThreshold=15
```

显示是否打印 GC 详细信息：

```
C:\Documents and Settings\Administrator>jinfo -flag PrintGCDetails  2972
-XX:-PrintGCDetails
```

除了查找参数的值，jinfo 也支持修改部分参数的数值，当然，这个修改能力是极其有限的。下例显示了通过 jinfo 对 PrintGCDetails 参数的修改，它可以在 Java 程序运行时，动态关闭或者打开这个开关。

```
C:\Documents and Settings\Administrator>jinfo -flag PrintGCDetails  2972
-XX:-PrintGCDetails

C:\Documents and Settings\Administrator>jinfo -flag +PrintGCDetails  2972

C:\Documents and Settings\Administrator>jinfo -flag PrintGCDetails  2972
-XX:+PrintGCDetails
```

注意：jinfo 不仅可以查看运行时某一个 Java 虚拟机参数的实际取值，甚至可以在运行时修改部分参数，并使之立即生效（并非所有参数都支持动态修改）。

6.3.4　导出堆到文件——jmap 命令

命令 jmap 是一个多功能的命令。它可以生成 Java 程序的堆 Dump 文件，也可以查看堆内对象实例的统计信息、查看 ClassLoader 的信息以及 finalizer 队列。

下例使用 jmap 生成 PID 为 2972 的 Java 程序的对象统计信息，并输出到 s.txt 文件中。

```
jmap -histo 2972 >c:\s.txt
```

输出文件有如下结构：

```
num     #instances         #bytes  class name
----------------------------------------------
  1:          4983        6057848  [I
  2:         20929        2473080  <constMethodKlass>
.................
1932:            1              8  sun.java2d.pipe.AlphaColorPipe
1933:            1              8  sun.reflect.GeneratedMethodAccessor64
Total       230478       22043360
```

可以看到，这个输出显示了内存中的实例数量和合计。

jmap 另一个更为重要的功能是得到 Java 程序的当前堆快照：

```
C:\Documents and Settings\Administrator>jmap -dump:format=b,file=c:\heap.hprof 2972
Dumping heap to C:\heap.hprof ...
Heap dump file created
```

本例中，将应用程序的堆快照输出到 C 盘 heap.bin 文件中。之后，可以通过多种工具分析该堆文件。比如，下文中提到的 jhat 工具，或者 Visual VM、MAT 等工具。

注意：jmap 可用于导出 Java 应用程序的堆快照。

此外，jmap 还可以查看系统的 ClassLoader 的信息，如下所示：

```
C:\Users\Administrator>jmap -permstat 10360
Attaching to process ID 10360, please wait...
class_loader classes bytes parent_loader  alive? type

<bootstrap> 368 1644840 null  live   <internal>
0x246f0b70  0   0       null       live   sun/misc/Launcher$ExtClassLoader@0x39114a50
0x246f94b8  9   63472   0x246f0b70 live   sun/misc/Launcher$AppClassLoader@0x39136bb8

total = 3      377    1708312    N/A       alive=3, dead=0    N/A
```

可以看到，系统中有 3 个 ClassCloader，并显示了它们的父子关系。同时，也显示了每个 ClassLoader 内部加载的类的数量和总大小。

通过 jmap，还可以观察系统 finalizer 队列中的对象，一个不恰当的 finalize()方法可能导致对象堆积在 finalizer 队列中。使用以下命令可以查看堆积在 finalizer 队列中的对象（第 5.5.6 节有更多详细的有关 finalizer 队列的信息）：

```
C:\Users\Administrator>jmap -finalizerinfo 9828
Number of objects pending for finalization: 13461

Count   Class description
```

```
------------------------------------------------------------
13458   geym.zbase.ch5.LongFinalize$LF
3      java.io.FileInputStream
```

可以看到，在 finalizer 队列中，堆积了大量的 geym.zbase.ch5.LongFinalize$LF 对象实例和 3 个 java.io.FileInputStream 实例。

6.3.5 JDK 自带的堆分析工具——jhat 命令

使用 jhat 工具可以用于分析 Java 应用程序的堆快照内容。

【示例 6-9】以前文中 jmap 的输出堆文件 heap.hprof 为例。

```
C:\Documents and Settings\Administrator>jhat c:\heap.hprof
Reading from c:\heap.hprof...
Dump file created Thu Mar 22 22:27:19 CST 2012
Snapshot read, resolving...
Resolving 133055 objects...
Chasing references, expect 26 dots..........................
Eliminating duplicate references..........................
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

jhat 在分析完成后，使用 HTTP 服务器展示其分析结果。在浏览器中访问 http://127.0.0.1:7000，结果如图 6.18 所示。

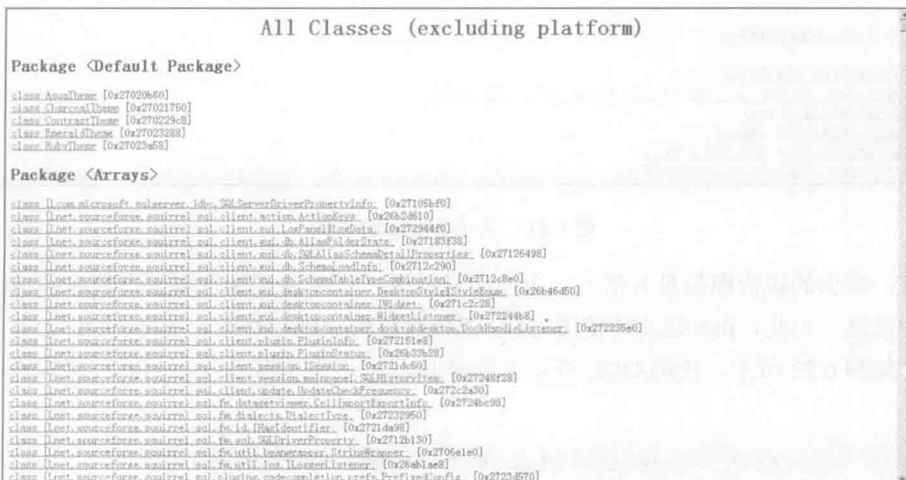


图 6.18　jhat 输出结果

在默认页中，jhat 服务器显示了所有的非平台类信息。点击链接进入，可以查看选中类的超类、ClassLoader 以及该类的实例等信息。此外，在页面底部，jhat 还为开发人员提供了其他查询方式，如图 6.19 所示。

Other Queries

- All classes including platform
- Show all members of the rootset
- Show instance counts for all classes (including platform)
- Show instance counts for all classes (excluding platform)
- Show heap histogram
- Show finalizer summary
- Execute Object Query Language (OQL) query

图 6.19 jhat 提供的查询功能

通过这些链接，开发者可以进一步查看所有类信息（包括 Java 平台的类）、所有类的实例数量以及实例的具体信息。最后，还有一个链接指向 OQL 查询界面。

图 6.20 显示了在 jhat 中，查看 Java 应用程序里 java.lang.String 类的实例数量。

```
21759 instances of class [C
19932 instances of class java.lang.String
8283 instances of class java.util.HashMap$Entry
5558 instances of class [B
```

图 6.20 显示实例数量

单击 instances 链接可以进一步查看 String 对象的实例，如图 6.21 所示。

Instances of 0x2b7dc2b8

class java.lang.String

com/sun/media/ (24 bytes)
Are you sure you want to delete the alias [0]? (24 bytes)
reqIncoming (24 bytes)
com/sun/corba/ (24 bytes)
javassist/compiler/ast (24 bytes)
menuitem.net.sourceforge.squirrel_sql.plugins.refactoring.actions.AddPrimaryKeyAction.accelerator (24 bytes)

图 6.21 显示 String 的实例

通常，导出的堆快照信息非常大，由于信息太多，可能很难通过页面上简单的链接索引找到想要的信息。为此，jhat 还支持使用 OQL 语句对堆快照进行查询。执行 OQL 语言的界面非常简洁，如图 6.22 所示。使用 OQL 查询出当前 Java 程序中所有 java.io.File 对象的路径。OQL 如下：

```
select file.path.value.toString() from java.io.File file
```

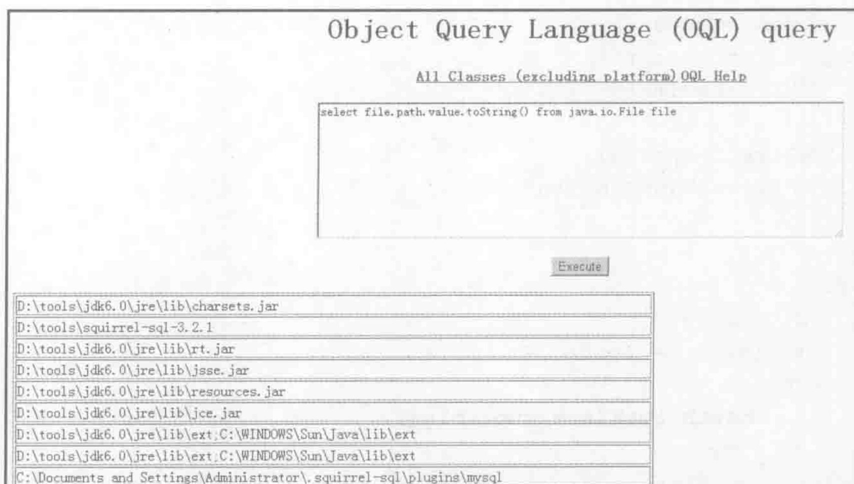


图 6.22　OQL 查询界面

jhat 的 OQL 语言与 Visual VM 的 OQL 非常接近，有兴趣的读者可以查阅本书中的相关章节。

注意：jhat 命令可以对堆快照文件进行分析，它启动一个 HTTP 服务器，开发人员可以通过浏览器，浏览 Java 堆快照。

6.3.6　查看线程堆栈——jstack 命令

jstack 可用于导出 Java 应用程序的线程堆栈，语法为：

```
jstack [-l] <pid>
```

-l 选项用于打印锁的附加信息。

jstack 工具会在控制台输出程序中所有的锁信息，可以使用重定向将输出保存到文件，如：

```
jstack -l 2348 >C:\deadlock.txt
```

【示例 6-10】下例演示了一个简单的死锁，两个线程分别占用 south 锁和 north 锁，并同时请求对方占用的锁，导致死锁发生。

```
public class DeadLock extends Thread{
 protected Object myDirect;
 static ReentrantLock south = new ReentrantLock();
 static ReentrantLock north = new ReentrantLock();

 public DeadLock(Object obj){
```

```
        this.myDirect=obj;
        if(myDirect==south){
            this.setName("south");
        }
        if(myDirect==north){
            this.setName("north");
        }
    }
    @Override
    public void run() {
        if (myDirect == south) {
            try {
                north.lockInterruptibly();              //占用 north
                try {
                    Thread.sleep(500);                  //等待 north 启动
                } catch (Exception e) {
                    e.printStackTrace();
                }
                south.lockInterruptibly();              //占用 south
                System.out.println("car to south has passed");
            } catch (InterruptedException e1) {
                System.out.println("car to south is killed");
            }finally{
                if(north.isHeldByCurrentThread())
                    north.unlock();
                if(south.isHeldByCurrentThread())
                    south.unlock();
            }

        }
        if (myDirect == north) {
            try {
                south.lockInterruptibly();              //占用 south
                try {
                    Thread.sleep(500);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                north.lockInterruptibly();                  //占用 north
                System.out.println("car to north has passed");
            } catch (InterruptedException e1) {
```

```
                System.out.println("car to north is killed");
            }finally{
                if(north.isHeldByCurrentThread())
                    north.unlock();
                if(south.isHeldByCurrentThread())
                    south.unlock();
            }

        }
    }

    public static void main(String[] args) throws InterruptedException {
        DeadLock car2south = new DeadLock(south);       //2 个线程死锁
        DeadLock car2north = new DeadLock(north);
        car2south.start();
        car2north.start();
        Thread.sleep(1000);
    }
}
```

使用 jstack 工具打印上例的输出，部分结果如下：

```
"north" prio=6 tid=0x0109c400 nid=0x17e4 waiting on condition [0x0517f000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x24704068> (a java.util.concurrent.locks.
ReentrantLock$NonfairSync)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt
(AbstractQueuedSynchronizer.java:834)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireInterruptibly
(AbstractQueuedSynchronizer.java:894)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireInterruptibly
(AbstractQueuedSynchronizer.java:1221)
        at java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.
java:340)
        at geym.zbase.ch6.hold.DeadLock.run(DeadLock.java:49)

   Locked ownable synchronizers:
        - <0x24704040> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)

"south" prio=6 tid=0x01061000 nid=0x12bc waiting on condition [0x04fdf000]
```

```
      java.lang.Thread.State: WAITING (parking)
          at sun.misc.Unsafe.park(Native Method)
          - parking to wait for  <0x24704040> (a java.util.concurrent.locks.
ReentrantLock$NonfairSync)
          at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
          at java.util.concurrent.locks.AbstractQueuedSynchronizer.
parkAndCheckInterrupt (AbstractQueuedSynchronizer.java:834)
          at java.util.concurrent.locks.AbstractQueuedSynchronizer.
doAcquireInterruptibly(AbstractQueuedSynchronizer.java:894)
          at java.util.concurrent.locks.AbstractQueuedSynchronizer.
acquireInterruptibly(AbstractQueuedSynchronizer.java:1221)
          at java.util.concurrent.locks.ReentrantLock.lockInterruptibly
(ReentrantLock.java:340)
          at geym.zbase.ch6.hold.DeadLock.run(DeadLock.java:29)

      Locked ownable synchronizers:
          - <0x24704068> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
   ....省略部分输出
   Found one Java-level deadlock:
   =============================
   "north":
     waiting for ownable synchronizer 0x24704068, (a java.util.concurrent.locks.
ReentrantLock$NonfairSync),
     which is held by "south"
   "south":
     waiting for ownable synchronizer 0x24704040, (a java.util.concurrent.locks.
ReentrantLock$NonfairSync),
     which is held by "north"

   ....省略部分输出
   Found 1 deadlock.
```

从 jstack 输出中，可以很容易地找到死锁的发生，并同时显示了发生死锁的两个线程，以及死锁线程的持有对象和等待对象，帮助开发人员解决死锁问题。

注意： 通过 jstack 工具不仅可以得到线程堆栈，它还能自动进行死锁检查，输出找到的死锁信息。

6.3.7 远程主机信息收集——jstatd 命令

在本节之前所述的工具中，只涉及到监控本机的 Java 应用程序，而在这些工具中，一些监

控工具也支持对远程计算机的监控（如 jps、jstat）。为了启用远程监控，则需要配合使用 jstatd 工具。

命令 jstatd 是一个 RMI 服务端程序，它的作用相当于代理服务器，建立本地计算机与远程监控工具的通信。jstatd 服务器将本机的 Java 应用程序信息传递到远程计算机，如图 6.23 所示。

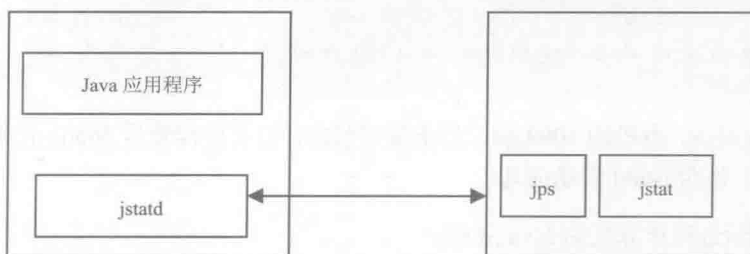


图 6.23　jstatd 工作示意图

直接打开 jstatd 服务器可能会抛出访问拒绝异常：

```
C:\Documents and Settings\Administrator>jstatd
Could not create remote object
access denied (java.util.PropertyPermission java.rmi.server.ignoreSubClasses write)
java.security.AccessControlException: access denied (java.util.PropertyPermission
java.rmi.server.ignoreSubClasses write)
      at
java.security.AccessControlContext.checkPermission(AccessControlContext.java:323)
      at java.security.AccessController.checkPermission(AccessController.java:546)
      at java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
      at java.lang.System.setProperty(System.java:725)
      at sun.tools.jstatd.Jstatd.main(Jstatd.java:122)
```

这是由于 jstatd 程序没有足够的权限所致。可以使用 Java 的安全策略，为其分配相应的权限，下面代码为 jstatd 分配了最大的权限，将其保存在 jstatd.all.policy 文件中：

```
grant codebase "file:E:/tools/jdk1.6.0.21/lib/tools.jar" {
 permission java.security.AllPermission;
};
```

然后，使用以下命令再次开启 jstatd 服务器：

```
C:\Documents and Settings\Administrator>jstatd -J-Djava.security.policy=
c:\jstatd.all.policy
```

服务器即可开启成功。

-J 参数是一个公共的参数，如 jps、jstat 等命令都可以接收这个参数。由于 jps、jstat 命令本身也是 Java 应用程序，-J 参数可以为 jps 等命令本身设置其 Java 虚拟机参数。

默认情况下，jstatd 将在 1099 端口开启 RMI 服务器。

```
C:\Documents and Settings\Administrator>netstat -ano|findstr 1099
 TCP    0.0.0.0:1099           0.0.0.0:0              LISTENING       3656
C:\Documents and Settings\Administrator>jps
3656 Jstatd
```

以上命令行显示，本机的 1099 端口处于监听状态，相关进程号是 3656。使用 jsp 显示 3656 进程正是 jstatd，说明 jstatd 启动成功。

使用 jps 显示远程计算机的 Java 进程：

```
C:\Documents and Settings\Administrator>jps localhost:1099
3656 Jstatd
460 Main
2464 Jps
844
```

使用 jstat 显示远程进程 460 的 gc 情况：

```
C:\Documents and Settings\Administrator>jstat -gcutil 460@localhost:1099
  S0     S1     E      O      P       YGC    YGCT    FGC    FGCT     GCT
  0.00  22.05  88.55  91.62  69.88     23    0.091     0    0.000    0.091
```

6.3.8 多功能命令行——jcmd 命令

在 JDK 1.7 以后，新增了一个命令行工具 jcmd。它是一个多功能的工具，可以用它来导出堆、查看 Java 进程、导出线程信息、执行 GC 等。

命令 jcmd 可以针对给定的 Java 虚拟机执行一条命令。首先来看一下使用 jcmd 列出当前系统中的所有 Java 虚拟机：

```
D:\tools\jdk1.7_40>jcmd -l
6192
7148 sun.tools.jcmd.JCmd -l
2828 geym.zbase.ch6.hold.DeadLock
```

参数-l 表示列出所有的 Java 虚拟机，针对每一个虚拟机，jcmd 可以使用 help 命令列出它们所支持的命令，如下所示：

```
D:\tools\jdk1.7_40>jcmd 2828 help
```

```
2828:
The following commands are available:
JFR.stop
JFR.start
JFR.dump
JFR.check
VM.native_memory
VM.commercial_features
ManagementAgent.stop
ManagementAgent.start_local
ManagementAgent.start
Thread.print
GC.class_histogram
GC.heap_dump
GC.run_finalization
GC.run
VM.uptime
VM.flags
VM.system_properties
VM.command_line
VM.version
Help
```

这里罗列的就是进程号 2828 虚拟机所支持的 jcmd 命令操作，这里介绍常用的几个命令。

（1）查看虚拟机启动时间 VM.uptime：

```
D:\tools\jdk1.7_40>jcmd 2828 VM.uptime
2828:
3805.305 s
```

命令 jcmd 也支持直接使用 MainClass 的名字来代替进程号，这样在编写脚本的时候也更为容易，如下命令：

```
D:\tools\jdk1.7_40>jcmd geym.zbase.ch6.hold.DeadLock VM.uptime
```

（2）打印线程栈信息：

```
D:\tools\jdk1.7_40>jcmd 2828 Thread.print
2828:
2014-10-07 10:46:09
Full thread dump Java HotSpot(TM) Client VM (24.0-b56 mixed mode, sharing):
省略所有线程信息
```

（3）查看系统中类的统计信息：

```
D:\tools\jdk1.7_40>jcmd 2828  GC.class_histogram
```

（4）导出堆信息：

```
D:\>jcmd 2828 GC.heap_dump D:\d.dump
2828:
Heap dump file created
```

GC.heap_dump 接收一个参数作为堆 Dump 文件的输出路径，得到堆文件后，可以使用 MAT 或者 Visual VM 等工具进行分析。

（5）获得系统的 Properties 内容：

```
D:\>jcmd 2828 VM.system_properties
2828:
#Tue Oct 07 13:12:56 CST 2014
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=D\:\\tools\\jdk1.7_40\\jre\\bin
java.vm.version=24.0-b56
java.vm.vendor=Oracle Corporation
….省略以下输出
```

（6）获得启动参数：

```
D:\>jcmd 2828 VM.flags
2828:
-XX:+FlightRecorder -XX:InitialHeapSize=16777216 -XX:MaxHeapSize=268435456
-XX:+UnlockCommercialFeatures -XX:-UseLargePagesIndividualAllocation
```

（7）获得所有性能统计相关数据。

通过 jcmd，还可以获得所有的 PerfData 数据，如下所示：

```
D:\>jcmd 2828 PerfCounter.print
2828:
java.cls.sharedLoadedClasses=704
java.cls.sharedUnloadedClasses=0
java.cls.unloadedClasses=83
...省略部分输出
java.threads.daemon=11
java.threads.live=14
java.threads.livePeak=16
java.threads.started=28
...省略部分输出
```

在这个输出中，可以找到所有的性能统计相关的数据，作为示例，这里给出了活动线程数量、活动线程峰值等信息。

从以上示例中可以看到，jcmd 拥有 jmap 的大部分功能，并且在 Oracle 的官方网站上也推荐使用 jcmd 命令代替 jmap。

6.3.9　性能统计工具——hprof

与前文中介绍的监控工具不同，hprof 不是独立的监控工具，它只是一个 Java agent 工具。它可以用于监控 Java 应用程序在运行时的 CPU 信息和堆信息。使用 java -agentlib:hprof=help 命令可以查看 hprof 的帮助文档。下面是 hropf 工具帮助信息的输出，加粗部分是常用的参数。

```
C:\Documents and Settings\Administrator>java -agentlib:hprof=help
    HPROF: Heap and CPU Profiling Agent (JVMTI Demonstration Code)
hprof usage: java -agentlib:hprof=[help]|[<option>=<value>, ...]
Option Name and Value   Description              Default
---------------------   -----------              -------
heap=dump|sites|all     heap profiling           all
cpu=samples|times|old   CPU usage                off
monitor=y|n             monitor contention       n
format=a|b              text(txt) or binary output a
file=<file>             write data to file       java.hprof[{.txt}]
net=<host>:<port>       send data over a socket  off
depth=<size>            stack trace depth        4
interval=<ms>           sample interval in ms    10
cutoff=<value>          output cutoff point      0.0001
lineno=y|n              line number in traces?   y
thread=y|n              thread in traces?        n
doe=y|n                 dump on exit?            y
msa=y|n                 Solaris micro state accounting n
force=y|n               force output to <file>   y
verbose=y|n             print messages about dumps y
```

使用 hprof 工具可以查看程序中各个函数的 CPU 占用时间。以下代码包含 3 个方法，分别占用不同的 CPU 时间。

```
public class HProfTest {
  public void slowMethod()
  {
    try {
      Thread.sleep( 1000 );                      //模拟一个很慢的方法
```

```
    } catch (InterruptedException e) {
      e.printStackTrace();     }
  }

  public void slowerMethod()
  {
    try {
      Thread.sleep( 10000 );                    //模拟一个更慢的方法
    } catch (InterruptedException e) {
      e.printStackTrace();     }
  }

  public void fastMethod()                      //一个很快的方法
  {
    try {
      Thread.yield();
    } catch (Exception e) {
      e.printStackTrace();     }
  }

  public static void main( String[] args )
  {
    HProfTest test = new HProfTest();
    test.fastMethod();                          //分别运行这些方法
    test.slowMethod();
    test.slowerMethod();
  }
}
```

使用参数-agentlib:hprof=cpu=times,interval=10 运行以上代码。times 选项将会在 Java 函数的调用前后记录函数的执行时间，进而计算函数的执行时间。在程序的当前目录下，会生成 java.hprof.txt 的文件，内部记录了性能统计信息，包含如下内容：

```
CPU TIME (ms) BEGIN (total = 11224) Tue Oct 07 13:36:52 2014
rank   self  accum   count trace method
   1 89.10% 89.10%       1 303832 geym.zbase.ch6.hold.HProfTest.slowerMethod
   2  8.91% 98.01%       1 301213 geym.zbase.ch6.hold.HProfTest.slowMethod
   3  0.12% 98.13%     114 303737 java.util.Properties.saveConvert
   4  0.08% 98.21%    3003 303730 java.lang.String.charAt
   5  0.08% 98.29%    3313 303732 java.lang.AbstractStringBuilder.append
   6  0.08% 98.37%    1172 300382 java.lang.Character.toLowerCase
   7  0.05% 98.42%    3313 303733 java.lang.StringBuffer.append
....省略部分内容
```

使用参数-agentlib:hprof=heap=dump,format=b,file=c:\core.hprof 运行程序，可以将应用程序的堆快照保存在指定文件 c:\core.hprof 中。使用 MAT 或者 Visual VM 等工具可以分析这个堆文件。

使用参数-agentlib:hprof=heap=sites 运行程序，可以输出 Java 应用程序中，每个跟踪点上的类所占内存的百分比，部分输出如下：

```
SITES BEGIN (ordered by live bytes) Tue Oct 07 13:39:48 2014
      percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs    bytes  objs trace name
  1  7.06%  7.06%     32800    2     32800    2 300254 char[]
  2  4.83% 11.89%     22416  174     22416  174 300006 char[]
  3 4.10%  15.98%     19040  595     19040   595 300177 java.util.LinkedHashMap$Entry
  4  3.53% 19.52%     16416    2    16416    2 300251 byte[]
```

这里以 rank 3 为例，说明在跟踪点 300177，分配了 4.83%的总内存，合计使用 22416 字节。在输出日志中查找 300177，可以得到部分堆栈为：

```
 TRACE 300177:
  java.util.HashMap$Entry.<init>(HashMap.java:814)
  java.util.LinkedHashMap$Entry.<init>(LinkedHashMap.java:325)
  java.util.LinkedHashMap.createEntry(LinkedHashMap.java:442)
  java.util.HashMap.addEntry(HashMap.java:888)
```

6.3.10　扩展 jps 命令

前文已经介绍过，包括 jps 在内的命令本质上是使用 Java 实现的。以 jps 命令为例，它在实现过程中，使用 MonitoredVmUtil 类，获得给定虚拟机的相关信息。而 MonitoredVmUtil 的主要功能都来自于 MonitoredVm.findByName()方法。它通过 PerfData 数据查询，可以得到虚拟机的相关性能参数。

【示例 6-11】下面的代码使用 MonitoredVm.findByName()方法查询了活动线程数、活动线程数峰值和 daemon 线程数。将这段代码加在 jps 的主实现中，就可以扩展 jps 的功能，在枚举系统 Java 进程的同时，也显示每一个 Java 进程的线程信息。

```
LongMonitor l= (LongMonitor)vm.findByName("java.threads.live");
output.append(" 活动线程: "+l.getValue()+" ");

LongMonitor str= (LongMonitor)vm.findByName("java.threads.livePeak");
output.append(" 活动线程峰值 g: "+str.getValue()+" ");

str= (LongMonitor)vm.findByName("java.threads.daemon");
```

```
output.append(" daemon 线程总数："+str.getValue()+"_");
```

运行修改后的 jps，显示如下：

```
6020 Jps 活动线程：5  活动线程峰值 g：5  daemon 线程总数：4
5572  活动线程：31  活动线程峰值 g：31  daemon 线程总数：22
6832 DeadLock 活动线程：9  活动线程峰值 g：9  daemon 线程总数：6
```

可以看到，除了显示进程 ID 和 MainClass 外，还显示了每一个进程的线程数量。

注意：本节所用到的类均在 OpenJDK 源码中，读者可以下载源码尝试修改 jps 工具的实现。

6.4 我是你的眼：图形化虚拟机监控工具 JConsole

JConsole 工具是 JDK 自带的图形化性能监控工具。通过 JConsole 工具，可以查看 Java 应用程序的运行概况，可以监控堆信息、永久区使用情况、类加载情况等。本节主要介绍 JConsole 工具的基本使用方法。

6.4.1 JConsole 连接 Java 程序

JConsole 程序在%JAVA_HOME%/bin 目录下，启动后，程序便要求指定连接 Java 应用程序，如图 6.24 所示。

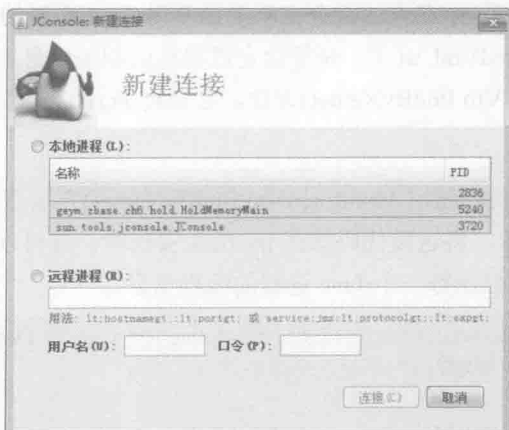


图 6.24　JConsole 连接 Java 应用

在新建连接对话框中，罗列了所有的本地 Java 应用程序。选择需要连接的程序即可。在本

地进程下，还有一个用于连接远程进程的文本框，输入正确的远程进程地址即可连接。

如果需要使用 JConsole 连接远程进程，则可以在远程 Java 应用程序启动时，加上如下参数：

```
-Djava.rmi.server.hostname=127.0.0.1
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=8888
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

-Djava.rmi.server.hostname 用于指定运行 Java 应用程序的计算机 IP 地址，-Dcom.sun.management.jmxremote.port 用于指定通过 JMX 管理该进程的端口号。基于以上配置启动的 Java 应用程序，通过 JConsole 在远程连接时，只需要填写如下远程进程即可：

```
127.0.0.1:8888
```

6.4.2　Java 程序概况

在连接上 Java 应用程序后，便可以查看应用程序概况，如图 6.25 所示。图中 4 张折线图分别显示了堆内存的使用情况、系统线程数量、加载类的数量以及 CPU 的使用率。

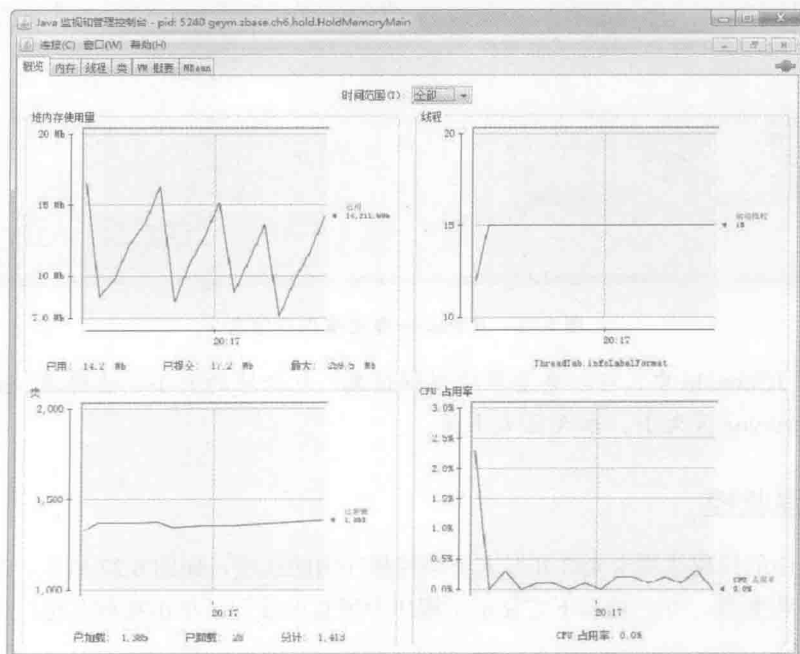


图 6.25　JConsole 查看应用程序概况

6.4.3 内存监控

切换到内存监控页面，JConsole 可以显示当前内存的详细信息。这不仅包括堆内存的整体信息，更细化到 eden 区、survivior 区、老年代的使用情况。同时，也包括非堆区，即永久代的使用情况。单击右上角的“执行 GC”按钮，可以强制应用程序进行一次 Full GC，如图 6.26 所示。

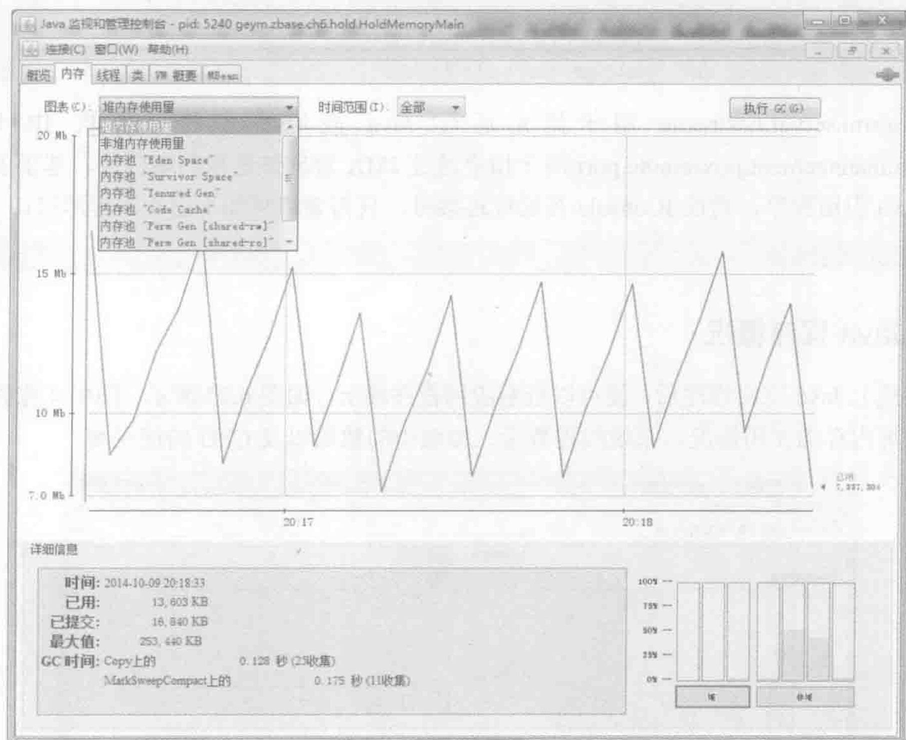


图 6.26 JConsole 查看堆内存信息

注意：在 JConsole 中，可以查看堆的详细信息，包括堆的大小、使用率、eden 区大小、survivor 区大小、永久区大小等。

6.4.4 线程监控

JConsole 中的线程选项卡允许开发人员监控程序内的线程，如图 6.27 所示。JConsole 显示了系统内的线程数量，并在屏幕下方显示了程序中所有的线程。单击线程名称，便可以查看线程的栈信息。

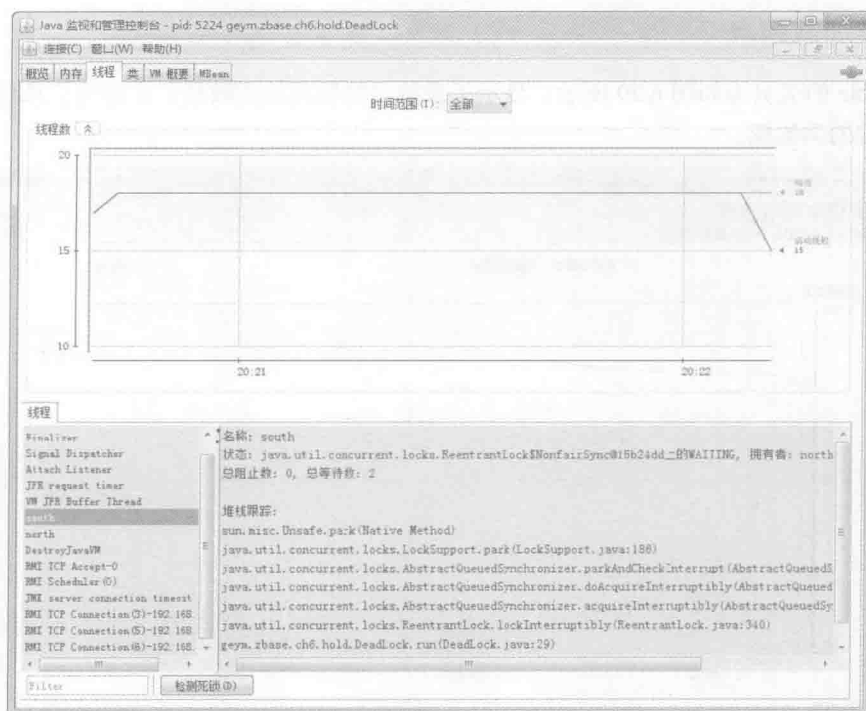


图 6.27　JConsole 查看线程信息

使用“检测到死锁”按钮，还可以自动检测多线程应用程序的死锁情况。图 6.28 展示了由 JConsole 检测到的死锁线程。

本例中使用的测试代码可以参考 6.3.6 节中的示例。

注意：使用 JConsole 可以方便地查看系统内的线程信息，并且可以快速地定位死锁问题。

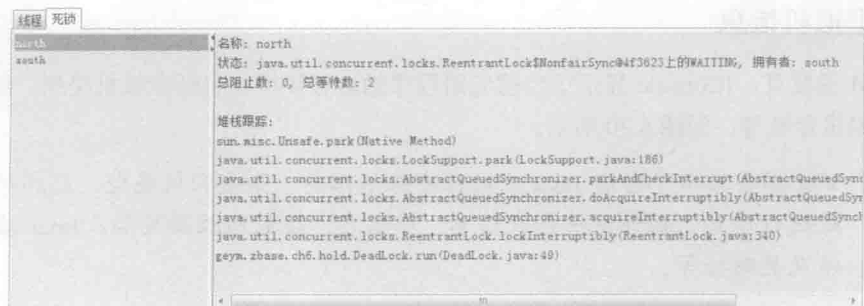


图 6.28　JConsole 死锁检查

6.4.5 类加载情况

JConsole 的类页面如图 6.29 所示，显示了系统已经装载的类数量。在详细信息栏中，还显示了已卸载的类数量。

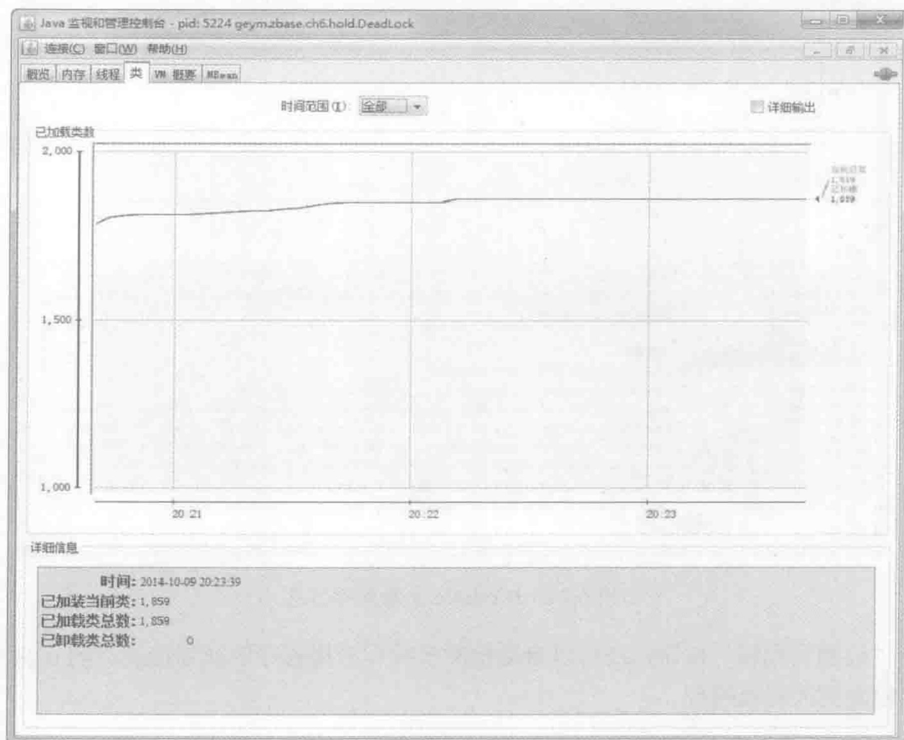


图 6.29　JConsole 类加载情况

6.4.6 虚拟机信息

在 VM 摘要页，JConsole 显示了当前应用程序的运行环境，包括虚拟机类型、版本、堆信息以及虚拟机参数等，如图 6.30 所示。

注意： VM 摘要显示了当前 Java 应用程序基本信息，如虚拟机类型、虚拟机版本、系统线程信息、操作系统内存信息、堆信息、垃圾回收器类型、Java 虚拟机参数以及类路径等。

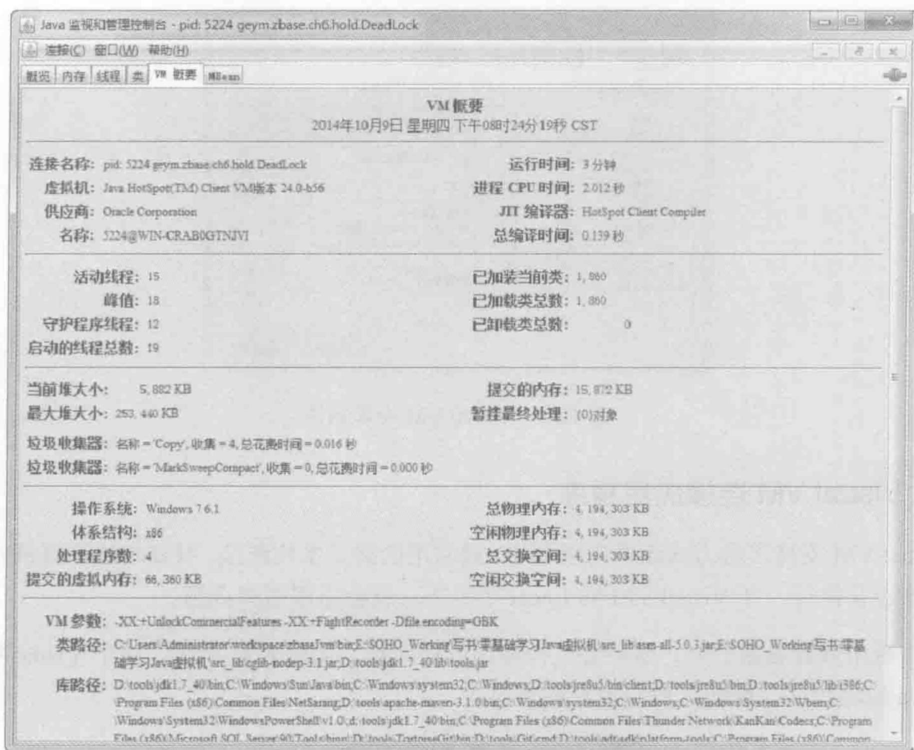


图 6.30　JConsole 展示 Java 虚拟机虚拟机信息

6.5　一目了然：可视化性能监控工具 Visual VM

Visual VM 是一个功能强大的多合一故障诊断和性能监控的可视化工具，它集成了多种性能统计工具的功能，使用 Visual VM 可以代替 jstat、jmap、jhat、jstack，甚至代替 JConsole。在 JDK 6 Update 7 以后，Visual VM 便作为 JDK 的一部分发布，即：它完全免费。

Viusal VM 也可以作为独立的软件安装，读者可以在 http://visualvm.java.net 下载并安装 Visual VM 的最新版本。

Visual VM 的一大特点是支持插件扩展，并且插件安装非常方便。我们既可以通过离线下载插件文件*.nbm，然后在 Plugin 对话框的已下载页面下，添加已下载的插件，也可以在可用插件页面下，在线安装插件，如图 6.31 所示。



图 6.31　Viusal VM 安装插件

6.5.1　Visual VM 连接应用程序

Visual VM 支持多种方式连接应用程序，最常用的就是本地连接。只要本地计算机内有 Java 应用程序正在执行，在 Visual VM 的 Local 节点下，就会出现这些应用。

双击应用或者右键打开，就能够监控应用程序运行，如图 6.32 所示。由于 Visual VM 本身也是 Java 应用程序，因此，自身也在列表内。

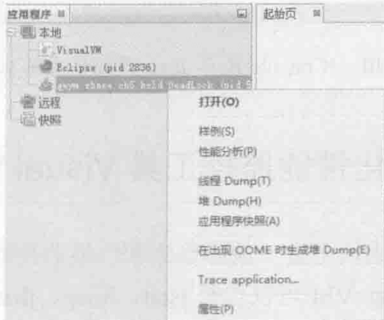


图 6.32　Viusal VM 监控应用程序

除了本地连接外，Visual VM 也支持远程 JMX 连接。Java 应用程序可以通过以下参数打开 JMX 端口：

```
-Djava.rmi.server.hostname=127.0.0.1
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=8888
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

通过如图 6.33 所示的操作来添加 JMX 连接。

在弹出的对话框中填写远程计算机地址、端口，如图 6.34 所示。如果需要验证，则再填写用户名和密码。

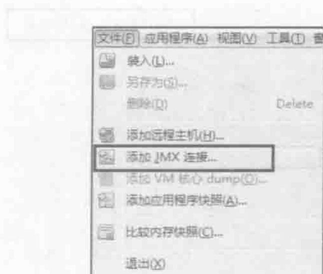


图 6.33　Viusal VM 添加 JMX 连接



图 6.34　Viusal VM 配置 JMX 连接

添加成功后，在 Local 节点下就会出现一个 JMX 图标的应用程序，如图 6.35 所示。图中两个应用程序分别通过本地方式与 JMX 方式连接，两者的标识图标是不同的。

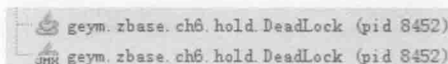


图 6.35　Viusal VM 中 JMX 应用程序显示

6.5.2　监控应用程序概况

通过 Visual VM，可以查看应用程序的基本情况，比如进程 ID、Main Class、启动参数等，如图 6.36 所示。



图 6.36　Viusal VM 显示应用程序概况

单击 Tab 页面上的监视页面，即可监控应用程序 CPU、堆、永久区、类加载和线程数的总体情况。通过页面上的“执行垃圾回收”和“堆 Dump”按钮还可以手工执行 Full GC 和生成堆快照，如图 6.37 所示。

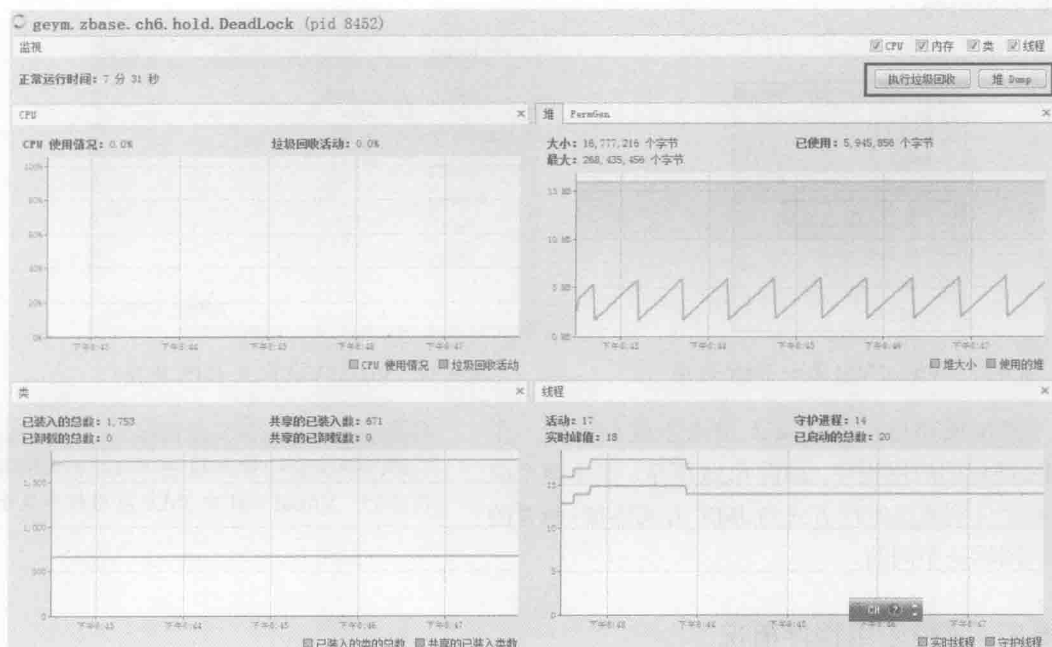


图 6.37　Viusal VM 显示 Java 程序内存信息

注意：Visual VM 将 CPU 使用率、堆信息、永久区信息、线程以及类加载情况作了可视化的展示，方便开发人员查看。

6.5.3　Thread Dump 和分析

Visual VM 的线程页面（如图 6.38 所示）可以提供详细的线程信息。单击右上角的“线程 Dump”按钮可以导出当前所有线程的堆栈信息（相当于 jstack）。

如果 Visual VM 在当前程序中找到死锁，则会以十分显眼的方式在线程页面给予提示，如图 6.39 所示。

注意：Visual VM 的 Thread 页面提供了详细的线程信息。在该页面，还会进行自动的死锁监测，一旦发现存在死锁现象便会提示用户。



图 6.38　Viusal VM 线程查看



图 6.39　Viusal VM 的死锁提示

6.5.4　性能分析

Visual VM 有两个采样器，在“Sampler”页面下，显示了 CPU 和内存两个性能采样器，用于实时地监控程序信息。CPU 采样器可以将 CPU 占用时间定位到方法，内存采样器可以查看当前程序的堆信息。

通过 Visual VM 的采样功能，可以找到该程序中占用 CPU 时间最长的方法，如图 6.40 所示。可以看到 HoldCPUTask.run()方法占用了大量的 CPU 时间，而相对的 LazyTask.run()方法就非常空闲。通常，可以根据这个功能，简单地定位到系统中最消耗资源的函数，并加以改进。

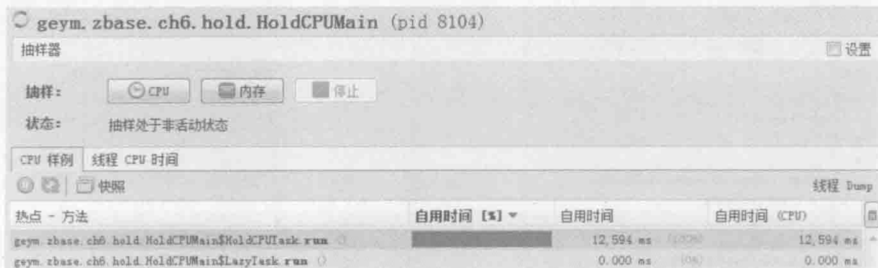


图 6.40　Viusal VM 方法耗时监控

在 Visual VM 的默认统计信息中，并不包含 JDK 的内置对象的函数调用统计，比如 java.*

包中的类。如果需要统计 JDK 内的方法调用情况，需要单击右上角的“设置”选项，手工进行配置。如图 6.41 所示，增加了对 java.util.*包内的函数调用统计信息。可以看到，java.util.Random.next()方法最为消耗 CPU 资源。

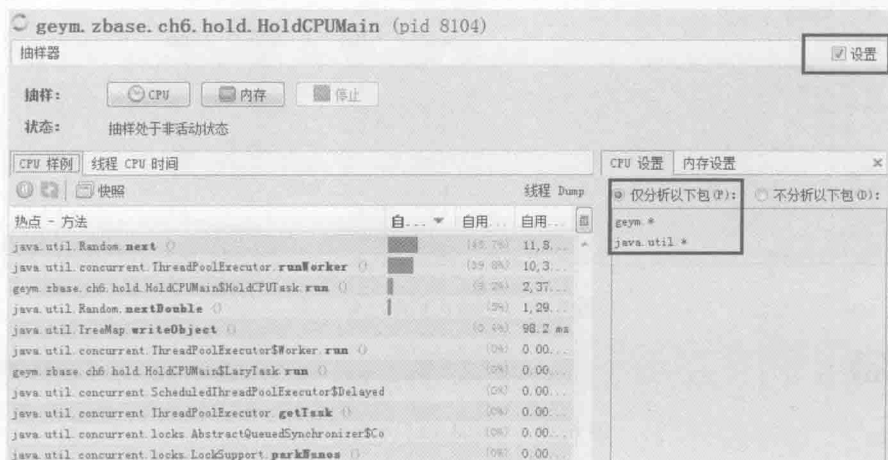


图 6.41 增加对 java.util.*的监控

通过内存采样器，可以实时查看系统中实例的分布情况，如图 6.42 所示。随着程序的运行，Visual VM 会实时更新这些数据，动态显示各个 class 所占的内存大小。

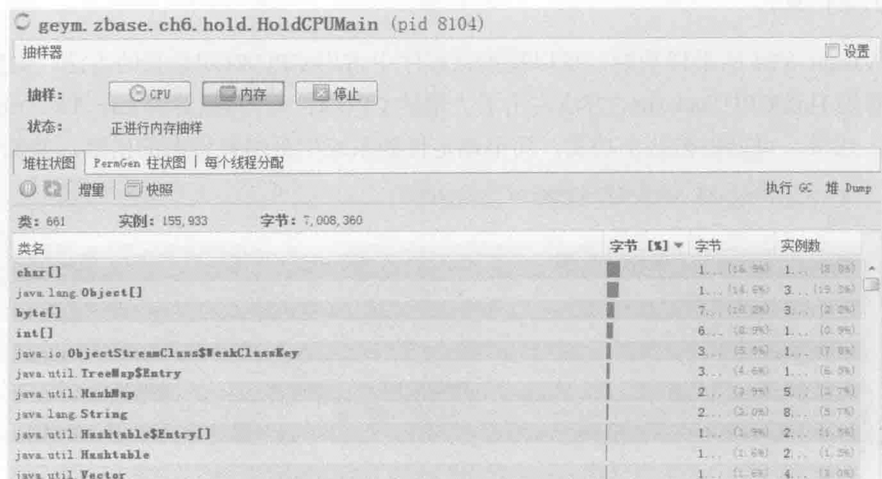


图 6.42 Viusal VM 内存监控

6.5.5　内存快照分析

通过右键菜单中的“堆 Dump”选项，可以立即获得当前应用程序的内存快照，如图 6.43 所示（相当于 jmap 命令）。

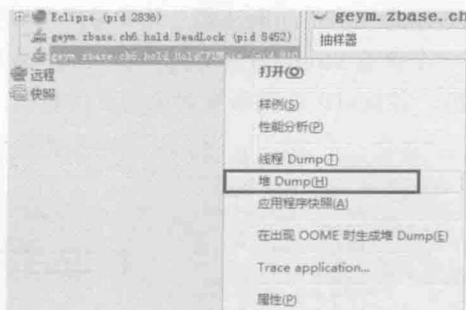


图 6.43　内存快照分析功能

内存快照分析功能如图 6.44 所示，在顶部的 Tab 页中，提供了 4 个基本功能页：概要、类、实例和 OQL 控制台。下面分别介绍它们。

（1）概要页面展示了当前内存的整体信息，包括内存大小、实例总数、类总数等。

（2）在类页面中，以类为索引，显示了每个类的实例数占用空间。在类页面中，还可以对两个不同的内存快照文件进行比较，这个功能可以帮助开发者快速分析同一应用程序在运行的不同时刻，内存数据产生的变化，如图 6.44 所示。



图 6.44　内存快照分析与快照比较

在这个类展示的页面中，如果需要获取指定类的更多信息，可以单击右键进入该类的实例页面，如图 6.45 所示。



图 6.45　进入类实例页面

（3）在实例页面中，将显示指定类的所有实例。开发者便可以查看当前内存中，内存数据的实际内容。图 6.46 显示了一个查看 String 对象实例页面的部分内容。可以看到，系统中所有的 String 对象都一一被罗列出，并且可以看到所有对象的具体数据。



图 6.46　String 对象实例显示

在右下角的引用页面中，可以查看到系统中引用这个实例的对象，进而展示对象间的引用关系。

（4）OQL 控制台提供了更为强大的对象查询功能。有关 Visual VM 的 OQL 支持，将在后续章节进行详细阐述。

注意：通过 Visual VM 提供的内存快照分析工具，可以查看堆快照内的类信息和对象信息。

6.5.6　BTrace 介绍

BTrace 是一款非常有意思的工具，它可以在不停机的情况下，通过字节码注入动态监控系统的运行情况，它可以跟踪指定的方法调用、构造函数调用和系统内存等信息。本节在参考 BTrace 用户手册的基础上，着眼于实际应用，选取较为实用的几个 BTrace 脚本，演示 BTrace 工具强大的功能。在 Visual VM 中安装 BTrace 的方法如图 6.47 所示。

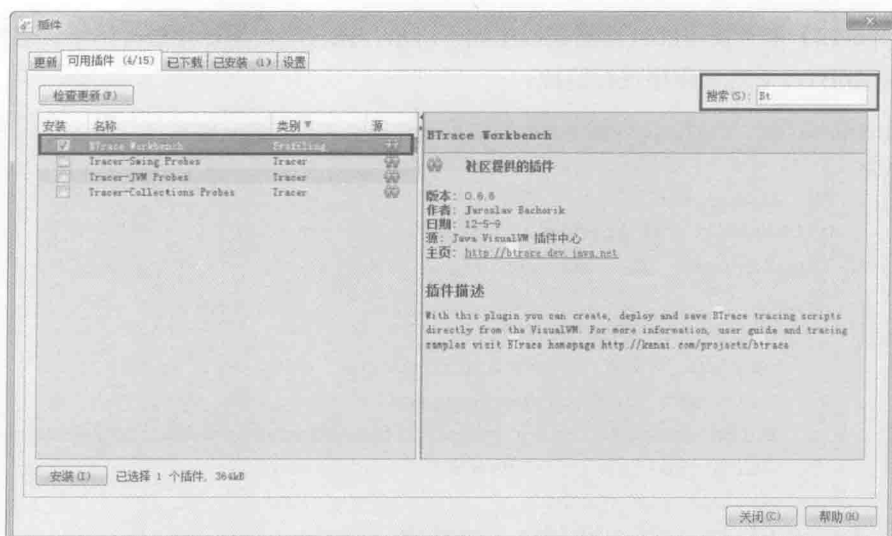


图 6.47　BTrace 的安装

在 Visual VM 中安装 BTrace 插件后，便可以针对 Java 应用程序执行 BTrace 脚本了。在 Java 应用程序节点上单击右键，选择“Trace application”即可进入 BTrace 插件界面，如图 6.48 所示。

BTrace 界面中主要分为上下两部分。上半部分为代码区，下半部分为输出结果。在代码区中完成 BTrace 脚本的编写，单击 Start 将 BTrace 脚本注入选中应用程序，程序输出结果将在下半部分显示，如图 6.49 所示。

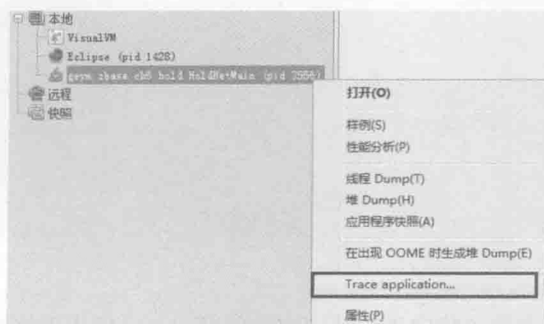


图 6.48　进入 BTrace 界面

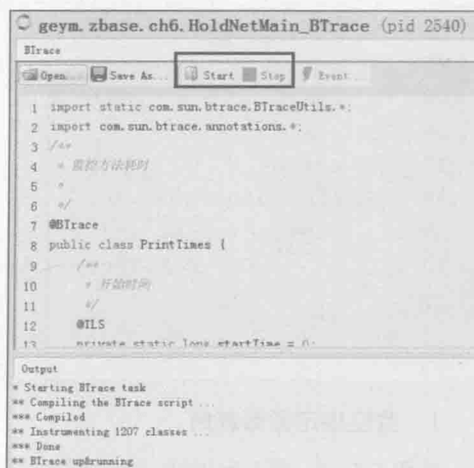


图 6.49　BTrace 插件运行界面

【示例 6-12】本节使用的目标测试程序如下所示，该程序周期性地进行网络读取操作。下文中，将使用 BTrace 对该程序进行监控。

```
public static class HoldNetTask implements Runnable {
    public void visitWeb(String strUrl){
        URL url = null;
        URLConnection urlcon = null;
        InputStream is = null;
        try {
            url = new URL(strUrl);
            urlcon = url.openConnection();
            is = urlcon.getInputStream();
            BufferedReader buffer = new BufferedReader(new InputStreamReader(is));
            StringBuffer bs = new StringBuffer();
            String l = null;
            while ((l = buffer.readLine()) != null) {
                bs.append(l).append("\r\n");
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                }
            }
        }
    }
    @Override
    public void run() {
        while (true) {
            visitWeb("http://www.sina.com.cn");
        }
    }
}
```

1. 监控指定函数耗时

使用 BTrace 脚本可以通过正则表达式，指定监控特定类的特定方法的耗时。以下代码将监控所有类中，名为 visitWeb ()的函数的执行时间。

```
import static com.sun.btrace.BTraceUtils.*;
import com.sun.btrace.annotations.*;
/**
 * 监控方法耗时
 *
 */
@BTrace
public class PrintTimes {
    /**
     * 开始时间
     */
    @TLS
    private static long startTime = 0;

    /**
     * 方法开始时调用
     */
    @OnMethod(clazz = "/.+/",                          //监控任意类
  method = "/visitWeb/")                               //监控 visitWeb ()方法
    public static void startMethod() {
        startTime = timeMillis();
    }

    /**
     * 方法结束时调用<br>
     * Kind.RETURN 这个注解很重要
     */
    @SuppressWarnings("deprecation")
    @OnMethod(clazz = "/.+/",
  method = "/visitWeb/",
  location = @Location(Kind.RETURN))                  //方法返回时触发
    public static void endMethod() {
        print(strcat(strcat(name(probeClass()), "."), probeMethod()));
        print(" [");
        print(strcat("Time taken : ", str(timeMillis() - startTime)));
        println("]");
    }
}
```

以上脚本使用@OnMethod 注释指定要监控的类和方法名称。它在函数开始运行时记录函数的起始执行时间，在函数返回时记录函数的终止时间，从而计算函数的运行耗时。将这段脚本

注入样例目标程序，部分输出如下：

```
geym.zbase.ch6.HoldNetMain_BTrace$HoldNetTask.visitWeb [Time taken : 1343]
geym.zbase.ch6.HoldNetMain_BTrace$HoldNetTask.visitWeb [Time taken : 1295]
geym.zbase.ch6.HoldNetMain_BTrace$HoldNetTask.visitWeb [Time taken : 1332]
geym.zbase.ch6.HoldNetMain_BTrace$HoldNetTask.visitWeb [Time taken : 1293]
```

注意：通过 BTrace 脚本，可以监控指定的某一个函数的运行耗时。

2. 监控函数参数

在上面的示例中，展示了通过 BTrace 监控 visitWeb()函数的执行时间。除了执行时间外，当前函数访问哪个网站依然是一个让人很关注的信息。通过 BTrace，可以很容易地获得 visitWeb()函数的参数信息。

```
import static com.sun.btrace.BTraceUtils.*;
import com.sun.btrace.annotations.*;
import com.sun.btrace.AnyType;
/**
 * 监控方法耗时
 *
 */
@BTrace
public class PrintTimes {
    @OnMethod(
  clazz="/.*HoldNetTask/",                             //要监控的类
  method="/visitWeb/"                                  //要监控的方法
  )
  public static void anyWriteFile(@ProbeClassName String pcn,
      @ProbeMethodName String pmn,
      AnyType[] args) {
      print(pcn);                                      //类名称
      print(".");
      print(pmn);                                      //方法名称
      printArray(args);                                //传递给方法的参数
  }
}
```

以上脚本监控 HoldNetTask 类的 visitWeb()方法。并在每次方法调用时，输出系统传递给该方法的参数。程序的部分输出如下：

```
geym.zbase.ch6.HoldNetMain_BTrace$HoldNetTask.visitWeb[http://www.sina.com.cn, ]
geym.zbase.ch6.HoldNetMain_BTrace$HoldNetTask.visitWeb[http://www.sina.com.cn, ]
```

加粗部分即 visitWeb ()方法的接收参数。通过 BTrace 在不影响系统正常运行的情况下，捕获了所有指定方法的调用以及传入参数。

注意：Btrace 可以跟踪指定方法的传入参数。获取方法的参数值，对于现场问题排查会很有帮助。

3. 取得任意行代码信息

通过 BTrace 脚本@Location 注释，可以指定程序运行到某一行代码时，触发某一行为。下例显示了通过 BTrace 脚本获取 HoldNetTask 类第 27 行代码的信息（当目标程序运行到第 27 行时，触发 BTrace 脚本）。

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
@BTrace
public class AllLines {
@OnMethod(
   clazz="/.*HoldNetTask/",                    //监控以 HoldNetTask 结尾的类
   location=@Location(value=Kind.LINE, line=27)      //指定 27 行触发
)
public static void online(@ProbeClassName String pcn,
   @ProbeMethodName String pmn,
   int line) {
   print(Strings.strcat(pcn, "."));
   print(Strings.strcat(pmn, ":"));
   println(line);
   }
}
```

以上脚本在目标程序运行到第 27 行时触发，并打印该行信息。该脚本部分输出如下：

```
geym.zbase.ch6.HoldNetMain_BTrace$HoldNetTask.visitWeb:27
geym.zbase.ch6.HoldNetMain_BTrace$HoldNetTask.visitWeb:27
```

由脚本输出可以看到，BTrace 正确识别出 HoldNetTask 类的第 27 行代码，正处于 visitWeb() 函数的运行区间中。若将@Location 中 line 的值设置为-1，则 BTrace 脚本将在每一代码行触发。

4. 定时触发

BTrace 脚本支持定时触发。可以周期性地执行某一行为，获取系统信息。下例使用@OnTimer 标记分别制定两个周期性任务，分别为每秒运行一次和每 3 秒运行一次。

```
import com.sun.btrace.annotations.*;
```

```
02 import static com.sun.btrace.BTraceUtils.*;
03
04 @BTrace
05 public class Timers {
06   @OnTimer(1000)                                           //每秒运行
07   public static void getUpTime() {
08       println(Strings.strcat("1000 msec: ",
09           Strings.str(Sys.VM.vmUptime())));                //虚拟机启动时间
10   }
11
12   @OnTimer(3000)                                           //每 3 秒运行
13   public static void getStack() {
14       jstackAll();                                         //导出线程堆栈
15   }
16 }
```

在脚本的第 6 行，指定一个每秒运行 1 次的任务，并打印虚拟机的启动时间。在第 12 行，指定一个每 3 秒运行一次的任务，每次都将导出系统的线程快照。

注意：通过 BTrace 可以在当前应用程序中，定时获取一些运行时系统信息。

5. 获取类的属性

在程序运行过程中，BTrace 还可以在指定的位置获得对象实例的字段信息。比如在本示例中，可以在 URL.openConnection()调用时，查看实际打开的 URL 地址，代码如下：

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
import java.lang.reflect.Field;

@BTrace
public class PrintField {
  //打印实例属性
  @OnMethod(clazz="/.*URL/",
      method = "/.*openConnection/", location = @Location(value = Kind.ENTRY))
  public static void visitWebEntry(@Self Object self ){
      println(strcat(strcat(name(probeClass()), "."), probeMethod()));
     println(self);
     println(get(field(classOf(self), "protocolPathProp")));  //只能取值 static 变量
      println(get(field(classOf(self), "host"),self));        //获得实例变量
     println("===========");
   }
}
```

这里监控 URL 对象，在调用 openConnection()方法时，获得当前对象实例 self，然后获取对应实例的 host 等属性。这段代码的执行结果如下：

```
java.net.URL.openConnection
http://www.sina.com.cn
java.protocol.handler.pkgs
www.sina.com.cn
```

6. 获取函数调用

BTrace 还可以获得一个方法的内部调用的方法信息。比如，本例中的 visitWeb()方法在其内部又调用了 URL.openConnection()、URLConnection.getInputStream()、BufferedReader.readLine()等方法。通过 BTrace，可以把这些调用的方法依次打印出来。

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
import java.lang.reflect.Field;

@BTrace
public class PrintField {
 @OnMethod(clazz="/.*HoldNetTask/",
                method = "visitWeb",location=@Location(value=Kind.CALL, clazz="/.*URL.*/", method="/.*/"))
 public static void visitWebEntry(@Self Object self, @ProbeClassName String pcm, @ProbeMethodName String pmn,
                    @TargetInstance Object instance, @TargetMethodOrField String method){
     println(Strings.strcat("Context: ", Strings.strcat(pcm, Strings.strcat("#", pmn))));
     print(classOf(instance));           //被调用目标对象
     print(".");
     println(Strings.strcat("",method));   //被调用方法
  }
}
```

上述代码监控 HoldNetTask. visitWeb()方法内部所调用到的包含“URL”为其类名的类的方法，并给出了被调用方法的名称。程序的输出如下：

```
Context: geym/zbase/ch6/HoldNetMain_BTrace$HoldNetTask#visitWeb
class java.net.URL.openConnection
Context: geym/zbase/ch6/HoldNetMain_BTrace$HoldNetTask#visitWeb
class sun.net.www.protocol.http.HttpURLConnection.getInputStream
```

可以看到，在 visitWeb()的执行过程中，先后执行了 openConnection()和 getInputStream()。在使用这种方式监控函数调用时，需要注意筛选被调用的方法，因为在一次函数执行中，可能会出现成百上千次函数调用，如果不加筛选，那么输出结果集可能会异常庞大而无法处理。

6.6 来自 JRockit 的礼物：虚拟机诊断工具 Mission Control

在 Oracle 收购 Sun 之前，Oracle 的 JRockit 虚拟机提供了一款叫做 JRockit Mission Control 的虚拟机诊断工具。在 Oracle 收购 Sun 之后，Oracle 公司同时拥有了 Sun Hotspot 和 JRockit 两款虚拟机。根据 Oracle 对于 Java 的战略，在今后的发展中，会将 JRockit 的优秀特性移植到 Hotspot 上。其中，一个重要的改进就是在 Sun 的 JDK 中加入了 JRockit 的支持。在 Oracle JDK 7 Update 40 之后，Mission Control 这款工具已经绑定在 Oracle JDK 中发布（很不幸，OpenJDK 中并没有该工具）。

Mission Control 位于%JAVA_HOME%/ bin/jmc.exe，打开这款软件，界面如图 6.50 所示。



图 6.50 Oracle Java Mission Control 主要界面

6.6.1 MBean 服务器

在左侧的 JVM 浏览器中，枚举了系统内所有的 Java 进程。打开其中一个进程的 MBean 服务器，可以看到如图 6.51 所示的界面。

可以看到，Mission Control 的界面非常有特色，在默认的界面中，以飞机仪表的视图显示了 Java 堆使用率、CPU 使用率和 Live Set+Fragmentation。

Mission Control 的一大特点是可以自由设置图标内容。比如，如果希望在飞机仪表面板再增加一个监控项，可以单击右侧的添加按钮，如图 6.52 所示。



图 6.51　Oracle Java Mission Control MBean 服务器



图 6.52　添加新的仪表盘

这里添加了一个线程数量作为示例，如图 6.53 所示。



图 6.53　添加了线程数量飞机仪表面板

除了飞机仪表面板外，Mission Control 的整个概览面板都是可以调整的。如图 6.54 所示，

在默认的折线图中，又添加了 eden 区的使用率。

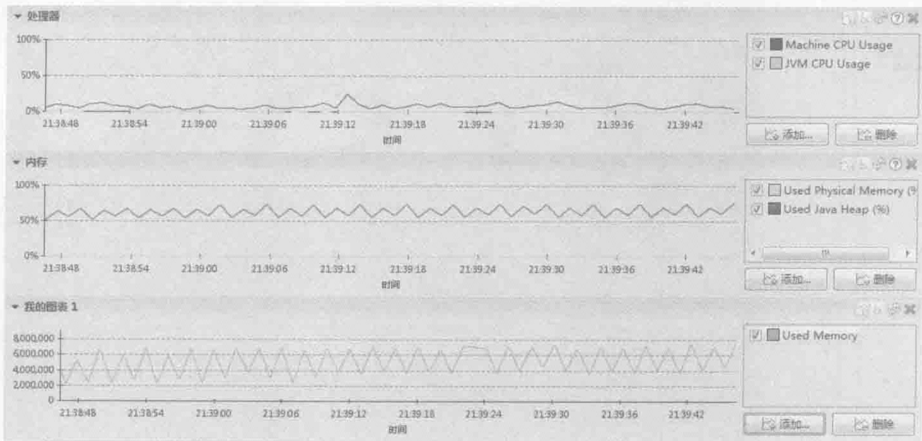


图 6.54　在折线图中添加新的性能选项

单击中部的按钮“运行时”，如图 6.55 所示，通过左下侧的 Tab 页切换，还可以显示内存、垃圾收集、内存池、线程等信息。

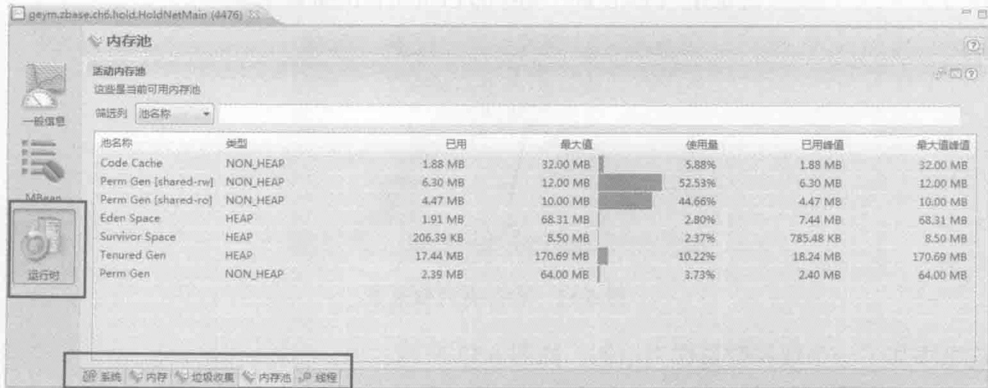


图 6.55　运行时信息

6.6.2　飞机记录器（Flight Recorder）

飞机记录器是 Mission Control 提供的另一大功能，和 MBean 服务器不同，它通过记录程序在一段时间内的运行情况，将记录结果进行分析和展示，可以更进一步对系统的性能进行分析和诊断。要使用飞行记录器，对于要监控的程序，必须带有以下参数启动：

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

从参数的命名上可以看到，这个功能带有很重的商业色彩。

如图 6.56 所示，在启动飞行记录器后，默认会进行 1 分钟的系统性能数据采集。

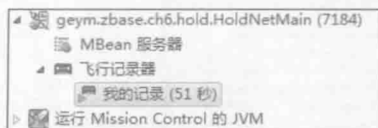


图 6.56　启动飞行记录器

记录结束后，Mission Control 会自动打开刚才的记录，如图 6.57 所示。可以看到，飞行记录器显示了更多的信息。

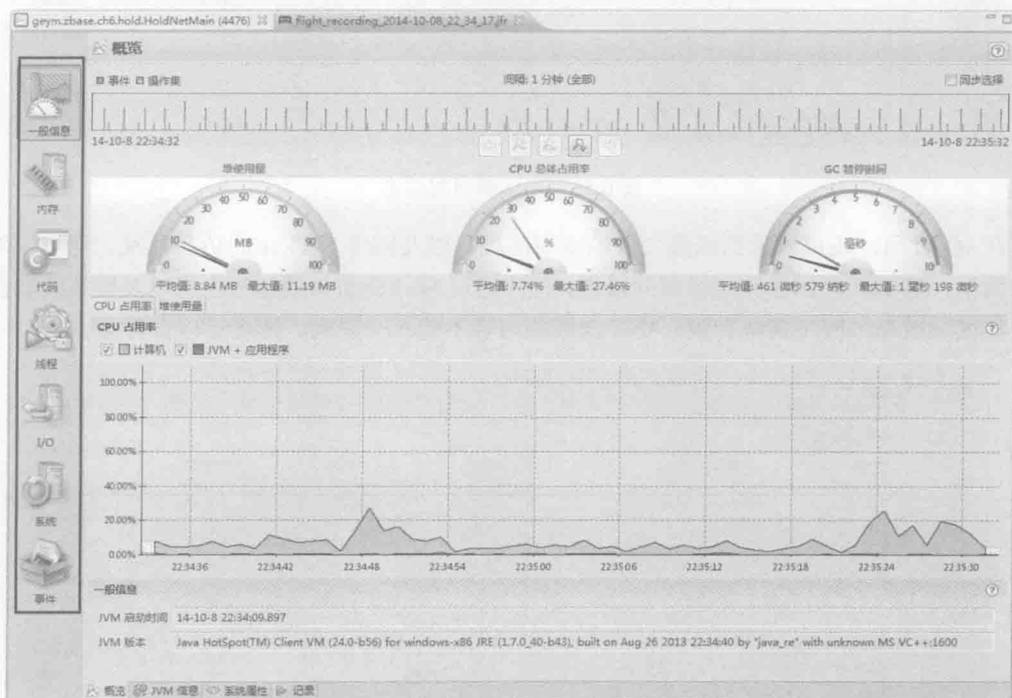


图 6.57　飞行记录器

除了常规的内存和 CPU 信息外，还有代码、线程和 IO 等比较重要的信息展示。以代码为例，它可以显示系统中的热点方法和占用的时间，如图 6.58 所示，显示了占用 CPU 时间最多的方法调用树信息。

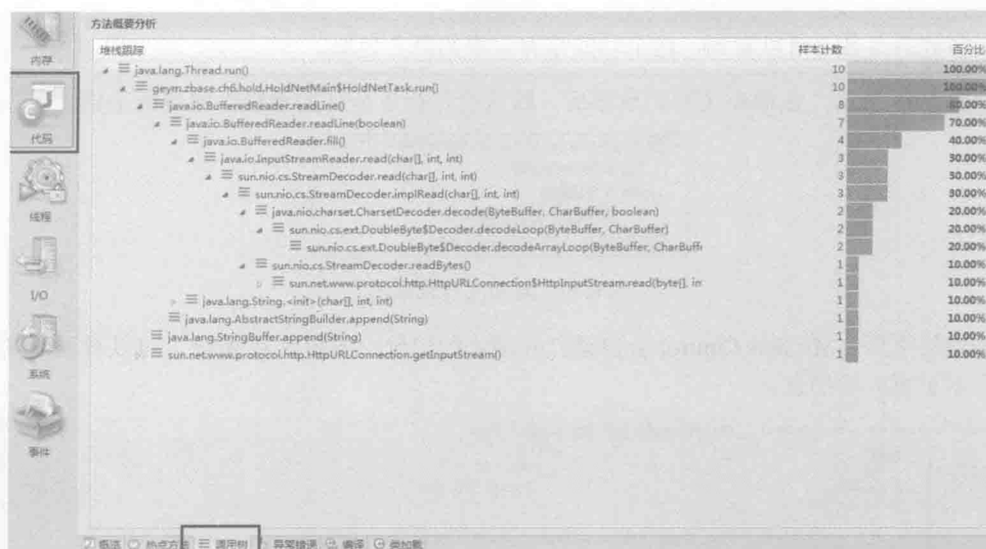


图 6.58　方法调用树信息

在 IO 页面，还可以看到磁盘文件的读写情况，以及网络 Socket 的访问情况。这里以网络访问为例，图 6.59 显示了在记录时间段内，程序通过 Socket 访问的远程主机以及数据读取次数和数据读取数量。限于篇幅有限，更多详细的内容不能一一罗列，读者可以自行尝试。

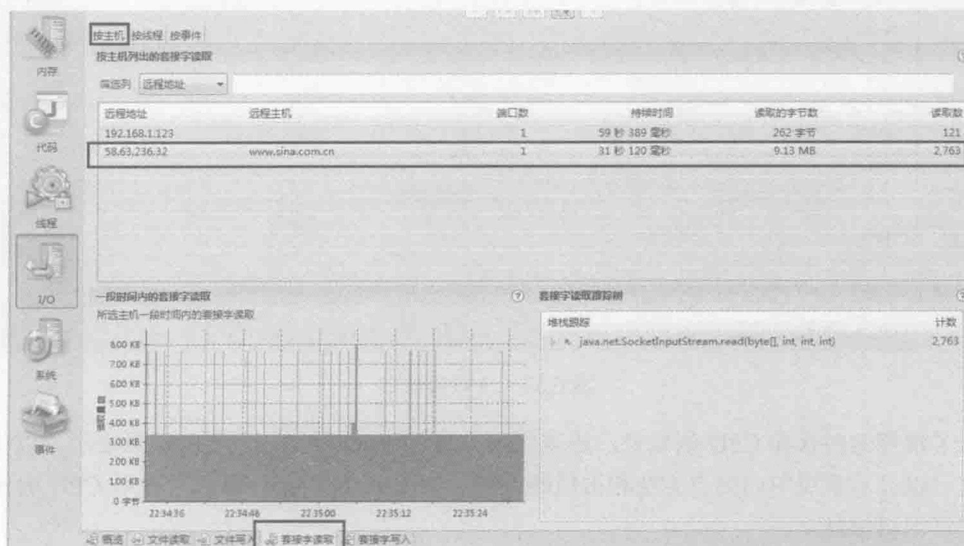


图 6.59　程序进行 Socket 读取的信息

6.7 小结

本章主要介绍了一些监控 Java 虚拟机和操作系统性能表现的实用工具。首先，在操作系统层面，主要涉及 Linux 和 Windows 两个平台，介绍了如何在系统中找到消耗资源最明显的进程和线程，进而辅助定位问题。其次，详细介绍了一些 JDK 自带的性能诊断工具，通过这些工具可以更细致地观察虚拟机的运行情况，同时，笔者抛砖引玉，给出一个加强和改进 jps 命令的方法，有兴趣的读者可以对此做更深入的研究，甚至开发自己的性能监控工具。

第 7 章 分析 Java 堆

内存一直是应用系统中最为重要的组成部分，在 Java 应用中，系统内存通常会被分为几块不同的空间，了解这些不同内存区域的作用有助于更好地编写 Java 应用，构建更加稳定的系统。而堆空间更是 Java 内存中最为重要的区域，几乎所有的应用程序对象都在堆中分配，当系统出现故障时，具备 Java 堆的内存分析能力，也可以更加方便地诊断系统的故障，而本章最主要的就是介绍有关 Java 堆的分析技术。

本章涉及的主要知识点有：

- 常见的内存溢出原因及其解决思路。
- 有关 java.lang.String 的探讨。
- 使用 Visual VM 分析堆。
- 使用 MAT 分析堆。

7.1 对症才能下药：找到内存溢出的原因

内存溢出（OutOfMemory，简称 OOM）是一个令人头痛的问题，它通常出现在某一块内存空间块耗尽的时候。在 Java 程序中，导致内存溢出的原因有很多，本节将主要讨论最常见的几种内存溢出问题，包括堆溢出、直接内存溢出、永久区溢出等。

7.1.1 堆溢出

堆是 Java 程序中最为重要的内存空间，由于大量的对象都直接分配在堆上，因此它也成为最有可能发生溢出的区间。一般来说，绝大部分 Java 的内存溢出都属于这种情况。其原因是因为大量对象占据了堆空间，而这些对象都持有强引用，导致无法回收，当对象大小之和大于由 Xmx 参数指定的堆空间大小时，溢出错误就自然而然地发生了。

【示例 7-1】下面这段代码就是堆溢出的典型，一个 ArrayList 对象总是持有 byte 数组的强引用，导致 byte 数据无法回收。

```
public class SimpleHeapOOM {
    public static void main(String args[]){
        ArrayList<byte[]> list=new ArrayList<byte[]>();
        for(int i=0;i<1024;i++){
            list.add(new byte[1024*1024]);
        }
    }
}
```

运行以上代码，应该会立即抛出错误：

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
 at geym.zbase.ch7.oom.SimpleHeapOOM.main(SimpleHeapOOM.java:14)
```

可以看到，在错误信息中注明了“java heap space”，表示这是一次堆空间的溢出。

为了缓解堆溢出错误，一方面可以使用-Xmx 参数指定一个更大的堆空间，另一方面，由于堆空间不可能无限增长，通过下文提到的 MAT 或者 Visual VM 等工具，分析找到大量占用堆空间的对象，并在应用程序上做出合理的优化也是十分必要的。

7.1.2 直接内存溢出

在 Java 的 NIO（New IO）中，支持直接内存的使用，也就是通过 Java 代码，获得一块堆外的内存空间，这块空间是直接向操作系统申请的。直接内存的申请速度一般要比堆内存慢，

但是其访问速度要快于堆内存。因此，对于那些可复用的，并且会被经常访问的空间，使用直接内存是可以提高系统性能的。但由于直接内存没有被 Java 虚拟机完全托管，若使用不当，也容易触发直接内存溢出，导致宕机。

【示例 7-2】下面的代码不断地申请直接内存，并最终可能导致内存溢出。

```
01 public class DirectBufferOOM {
02     public static void main(String args[]){
03         for(int i=0;i<1024;i++){
04             ByteBuffer.allocateDirect(1024*1024);
05             System.out.println(i);
06 //              System.gc();
07         }
08     }
09 }
```

注意代码第 6 行，System.gc()暂时被注释掉，也就是不会显式触发 GC。接着，在 Windows 平台上，使用 32 位 Java 虚拟机，根据以下参数运行上述代码：

```
-Xmx1g -XX:+PrintGCDetails
```

不用多久，程序就会因为内存溢出而退出，部分打印信息如下：

```
732
733
Exception in thread "main" java.lang.OutOfMemoryError
 at sun.misc.Unsafe.allocateMemory(Native Method)
 at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:127)
 at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:306)
 at geym.zbase.ch7.oom.DirectBufferOOM.main(DirectBufferOOM.java:14)
```

可以看到，在大约 733 次循环时，发生 OutOfMemoryError 错误。从堆栈可以看到，发生 OOM 时，正在进行 DirectByteBuffer 的分配。

提醒： 笔者在此使用的是 JDK 1.7u40 32 位 Java 虚拟机，如果使用 JDK 1.7 64 位虚拟机，程序是可以正常执行的，且不会出现 OOM，这是因为 32 位计算机系统对应用程序的可用最大内存有限制。以 Windows 平台为例，在 32 位系统中，进程的寻址空间为 4GB，其中 2GB 为用户空间，2GB 为系统空间，故实际可用的系统内存只有 2GB，当 Java 进程的所有内存之和（堆空间、栈空间、直接内存以及虚拟机自身所用的内存）大于 2GB 时，就会出现 OOM 的错误。

读者也许还会有一个疑问，就是在这里为什么 Java 的垃圾回收机制没有发挥作用？程序第

4 行分配的直接内存并没有被任何对象所引用，为何没有被回收呢？从程序的输入日志中也可以看到，虽然打开了-XX:+PrintGCDetails 开关，但是并没有一次 GC 日志，这说明在整个执行过程中，GC 并没有进行。事实上，直接内存不一定能够触发 GC（除非直接内存使用量达到了-XX:MaxDirectMemorySize 的设置），所以保证直接内存不溢出的方法是合理地进行 Full GC 的执行，或者设定一个系统实际可达的-XX:MaxDirectMemorySize 值（默认情况下等于-Xmx 的设置）。因此，如果系统的堆内存少有 GC 发生，而直接内存申请频繁，会比较容易导致直接内存溢出（这个问题在 32 位虚拟机上尤为明显）。

如果将上述代码中第 6 行的 System.gc()的注释去掉，使显式 GC 生效，那么程序将可以正常结束，这说明 GC 可以回收直接内存。

另一个让该程序正常执行的方法是设置一个较小的堆，在不指定-XX:MaxDirectMemorySize 的情况下，最大可用直接内存等于-Xmx 的值。

```
-Xmx512m -XX:+PrintGCDetails
```

这里将最大堆限制在 512MB，而非 1GB，这种情况下，最大可用直接内存也为 512MB，操作系统可以同时为堆和直接内存提供足够的空间，当直接内存使用量达到 512MB 时，也会进行 GC 释放无用内存空间。

此外，显式设置-XX:MaxDirectMemorySize 也是解决这一问题的方法。只要设置一个系统实际可达的最大直接内存值，那么像这种实际上不应该触发的内存溢出就不会发生了。

综上所述，为避免直接内存溢出，在确保空间不浪费的基础上，合理得执行显式 GC，可以降低直接内存溢出的概率，设置合理的-XX:MaxDirectMemorySize 也可以避免意外的内存溢出发生，而设置一个较小的堆在 32 位虚拟机上可以使得更多的内存用于直接内存。

7.1.3　过多线程导致 OOM

由于每一个线程的开启都要占用系统内存，因此当线程数量太多时，也有可能导致 OOM。由于线程的栈空间也是在堆外分配的，因此和直接内存非常相似，如果想让系统支持更多的线程，那么应该使用一个较小的堆空间。

【示例 7-3】下面的代码对这种情况作了演示，这里使用的依然是 Windows 平台 32 位 Java 虚拟机 JDK 1.7u40。

```
public class MultiThreadOOM {
    public static class SleepThread implements Runnable{
        public void run(){
```

```
            try {
                Thread.sleep(10000000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String args[]){
        for(int i=0;i<1500;i++){
            new Thread(new SleepThread(),"Thread"+i).start();
            System.out.println("Thread"+i+" created");
        }
    }
}
```

上述代码试图创建 1500 个 Java 线程，使用以下参数执行这个程序：

```
-Xmx1g
```

运行结果如下：

```
Thread1125 created
Thread1126 created
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:693)
    at geym.zbase.ch7.oom.MultiThreadOOM.main(MultiThreadOOM.java:23)
```

可以看到，在线程 1126 处，系统抛出了 OOM，并且打印出了"unable to create new native thread"，表示系统创建线程的数量已经饱和，其原因是 Java 进程已经达到了可使用的内存上限。要解决这个问题，也可以从以下两方面下手：

（1）一个方法是可以尝试减少堆空间，如使用以下参数运行程序：

```
-Xmx512m
```

使用 512MB 堆空间后，操作系统就可以预留更多内存用于线程创建，因此程序可以正常执行。

（2）另一个方法是减少每一个线程所占的内存空间，使用-Xss 参数可以指定线程的栈空间。尝试以下参数：

```
-Xmx1g -Xss128k
```

这里依然使用 1GB 的堆空间，但是将线程的栈空间减少到 128KB，剩余可用的内存理应可以容纳更多的线程，因此程序也可以正常执行。

注意：如果减少了线程的栈空间大小，那么栈溢出的风险会相应地上升。

因此，处理这类 OOM 的思路，除了合理的减少线程总数外，减少最大堆空间、减少线程的栈空间也是可行的。

7.1.4　永久区溢出

永久区（Perm）是存放类元数据的区域。如果一个系统定了太多的类型，那么永久区是有可能溢出的。在 JDK 1.8 中，永久区被一块称为元数据的区域替代，但是它们的功能是类似的，都是为了保存类的元信息。

【示例 7-4】如果一个系统不断地产生新的类，而没有回收，那最终非常有可能导致永久区溢出。下面这段代码每次循环都生成一个新的类（注意是类，而不是对象实例）。

```
public class PermOOM {
    public static void main(String[] args) {
     try{
        for(int i=0;i<100000;i++){
            CglibBean bean = new CglibBean("geym.jvm.ch3.perm.bean"+i,new
HashMap());
        }
     }catch(Error e){
          e.printStackTrace();
     }
    }
}
```

这里使用 JDK 1.6，并使用下述参数执行上述代码：

```
-XX:MaxPermSize=5m
```

程序运行一段时间后，抛出以下异常：

```
Caused by: java.lang.OutOfMemoryError: PermGen space
 at java.lang.ClassLoader.defineClass1(Native Method)
 at java.lang.ClassLoader.defineClassCond(ClassLoader.java:631)
 at java.lang.ClassLoader.defineClass(ClassLoader.java:615)
 ... 9 more
```

一般来说，要解决永久区溢出问题，可以从以下几个方面考虑：

- 增加 MaxPermSize 的值。
- 减少系统需要的类的数量。
- 使用 ClassLoader 合理地装载各个类，并定期进行回收。

7.1.5 GC 效率低下引起的 OOM

GC 是内存回收的关键，如果 GC 效率低下，那么系统的性能会受到严重的影响。如果系统的堆空间太小，那么 GC 所占的时间就会较多，并且回收所释放的内存就会较少。根据 GC 占用的系统时间，以及释放内存的大小，虚拟机会评估 GC 的效率，一旦虚拟机认为 GC 的效率过低，就有可能直接抛出 OOM 异常。但是，虚拟机不会对这个判定太随意，因为即使 GC 效率不高，强制中止程序还是显得有些太野蛮。一般情况下，虚拟机会检查以下几种情况：

- 花在 GC 上的时间是否超过了 98%。
- 老年代释放的内存是否小于 2%。
- eden 区释放的内存是否小于 2%。
- 是否连续最近 5 次 GC 都出现了上述几种情况（注意是同时出现，不是出现一个）。

只有满足所有条件，虚拟机才有可能抛出如下 OOM：

```
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

尽管虚拟机限制的条件如此严格，但是在绝大部分场合，还是会抛出堆溢出错误。由于这个 OOM 只是起到辅助作用，帮助提示系统分配的堆可能太小，因此虚拟机并不强制一定要开启这个错误提示，可以通过关闭开关-XX:-UseGCOverheadLimit 来禁止这种 OOM 的产生。

7.2 无处不在的字符串：String 在虚拟机中的实现

String 字符串一直是各种编程语言的核心。字符串应用之广泛，使得每一种计算机语言都必须对其做特殊的优化和实现。在 Java 中，String 虽然不是基本数据类型，但是也享有了和基本数据类型一样的待遇。本节将主要讨论字符串在虚拟机中的实现。

7.2.1 String 对象的特点

在 Java 语言中，Java 的设计者对 String 对象进行了大量的优化，其主要表现在以下 3 个方面，同时这也是 String 对象的 3 个基本特点：

- 不变性。
- 针对常量池的优化。
- 类的 final 定义。

1. 不变性

不变性是指 String 对象一旦生成，则不能再对它进行改变。String 的这个特性可以泛化成不变（immutable）模式，即一个对象的状态在对象被创建之后就不再发生变化。不变模式的主要作用在于，当一个对象需要被多线程共享，并且访问频繁时，可以省略同步和锁等待的时间，从而大幅提高系统性能。

> **注意：** 不变性可以提高多线程访问的性能。因为对象不可变，因此对于所有线程都是只读的，多线程访问时，即使不加同步也不会产生数据的不一致，故减小了系统开销。

由于不变性，一些看起来像是修改的操作，实际上都是依靠产生新的字符串实现的。比如 String.substring()、String.concat()方法，它们都没有修改原始字符串，而是产生了一个新的字符串，这一点是非常值得注意的。如果需要一个可以修改的字符串，那么需要使用 StringBuffer 或者 StringBuilder 对象。

2. 针对常量池的优化

针对常量池的优化指当两个 String 对象拥有相同的值时，它们只引用常量池中的同一个拷贝。当同一个字符串反复出现时，这个技术可以大幅度节省内存空间。

```
String str1=new String("abc");
String str2=new String("abc");
System.out.println(str1==str2);                    //返回 false
System.out.println(str1==str2.intern());           //返回 false
System.out.println("abc"==str2.intern());          //返回 true
```

以上代码 str1 和 str2 都开辟了一块堆空间存放 String 实例，如图 7.1 所示。虽然 str1 和 str2 内容相同，但是在堆中的引用是不同的。String.intern()返回字符串在常量池中的引用，显然它和 str1 也是不同的，但是，根据最后一行代码可以看到，String.intern()始终和常量字符串相等，读者可以思考一下，str1.intern()与 str2.intern()是否相等呢？

3. 类的 final 定义

除以上两点外，final 类型定义也是 String 对象的重要特点。作为 final 类的 String 对象在系统中不可能有任何子类，这是对系统安全性的保护。同时，在 JDK 1.5 版本之前的环境中，使

用 final 定义有助于帮助虚拟机寻找机会，内联所有的 final 方法，从而提高系统效率。但这种优化方法在 JDK 1.5 以后，效果并不明显。

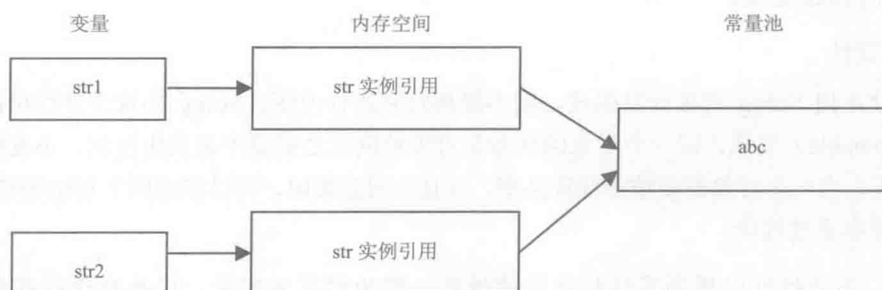


图 7.1　String 内存分配方式

7.2.2　有关 String 的内存泄漏

什么是内存泄漏？所谓内存泄漏，简单地说，就是由于疏忽或错误造成程序未能释放已经不再使用的内存的情况，它并不是说物理内存消失了，而是指由于不再使用的对象占据内存不被释放，而导致可用内存不断减小，最终有可能导致内存溢出。

由于垃圾回收器的出现，与传统的 C/C++相比，Java 已经把内存泄漏的概率大大降低了，所有不再使用的对象会由系统自动收集，但这并不意味着已经没有内存泄漏的可能。内存泄漏实际上更是一个应用问题，这里以 String.substring()方法为例，说明这种内存泄漏的问题。

在 JDK 1.6 中，java.lang.String 主要由 3 部分组成：代表字符数组的 value、偏移量 offset 和长度 count，如图 7.2 所示。

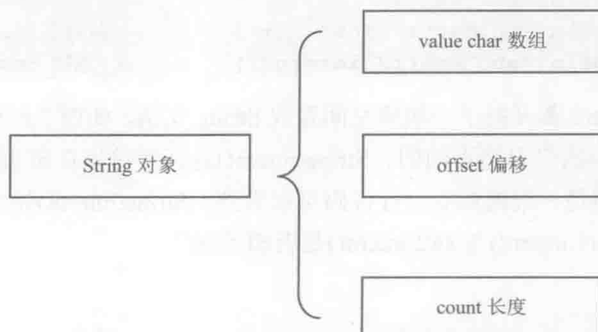


图 7.2　JDK 1.6 中 String 对象内部结构

这个结构为内存泄漏埋下了伏笔，字符串的实际内容由 value、offset 和 count 三者共同决定，而非 value 一项。试想，如果字符串 value 数组包含 100 个字符，而 count 长度只有 1 个字节，那么这个 String 实际上只有 1 个字符，却占据了至少 100 个字节，那剩余的 99 个就属于泄漏的部分，它们不会被使用，不会被释放，却长期占用内存，直到字符串本身被回收。如图 7.3 所示，显示了这种糟糕的情况。可以看到，str 的 count 为 1，而它的实际取值为字符串“0”，但是在 value 的部分，却包含了上万个字节，在这个极端情况中，原本只应该占用 1 个字节的 String，却占用了上万个字节，因此，可以判定为内存泄漏。

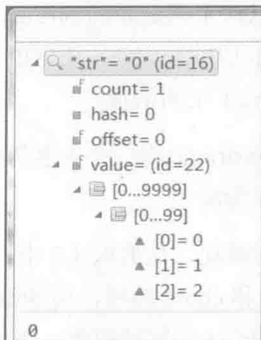


图 7.3　一个泄漏的 String

不幸的是，这种情况在 JDK 1.6 中非常容易出现。使用 String.substring()方法就可以很容易地构造这么一个字符串。下面简单解读一下 JDK 1.6 中 String.substring()的实现。

```
public String substring(int beginIndex, int endIndex) {
if (beginIndex < 0) {
    throw new StringIndexOutOfBoundsException(beginIndex);
}
if (endIndex > count) {
    throw new StringIndexOutOfBoundsException(endIndex);
}
if (beginIndex > endIndex) {
    throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
}
return ((beginIndex == 0) && (endIndex == count)) ? this :
    new String(offset + beginIndex, endIndex - beginIndex, value);
}
```

可以看到，在 substring()的实现中，最终是使用了 String 的构造函数，生成了一个新的 String。该构造函数的实现如下：

```
// Package private constructor which shares value array for speed.
String(int offset, int count, char value[]) {
this.value = value;
this.offset = offset;
this.count = count;
}
```

该构造函数并非公有构造函数，这点应该万幸，因为正是这个构造函数引起了内存泄漏问题。新生成的 String 并没有从 value 中获取自己需要的那部分，而是简单地使用了相同的 value 引用，只是修改了 offset 和 count，以此来确定新的 String 对象的值。当原始字符串没有被回收时，这种情况是没有问题的，并且通过共用 value，还可以节省一部分内存，但是一旦原始字符串被回收，value 中多余的部分就造成了空间浪费。

综上所述，如果使用了 String.substring()将一个大字符串切割为小字符串，当大字符串被回收时，小字符串的存在就会引起内存泄漏。

所幸，这个问题已经引起官方的重视，在 JDK 1.7 中，对 String 的实现有了大幅度的调整。在新版本的 String 中，去掉了 offset 和 count 两项，而 String 的实质性内容仅仅由 value 决定，而 value 数组本身也就代表了这个 String 实际的取值。下面，简单地对比 String.length()方法来说明这个问题，代码如下：

```
//JDK 1.7 的实现
public int length() {
    return value.length;
}

//JDK 1.6 的实现
public int length() {
    return count;
}
```

可以看到，在 JDK 1.6 中，String 的长度和 value 无关。基于这种改进的实现，substring()方法的内存泄漏问题也得以解决，如下代码所示，展示了 JDK 1.7 中的 String.substring()实现。

```
public String substring(int beginIndex, int endIndex) {
    //省略部分无关内容，读者自行查看代码
    int subLen = endIndex - beginIndex;
    //省略部分无关内容，读者自行查看代码
    return ((beginIndex == 0) && (endIndex == value.length)) ? this
            : new String(value, beginIndex, subLen);
}
```

```
public String(char value[], int offset, int count) {
    //省略部分无关内容，读者自行查看代码
    // Note: offset or count might be near -1>>>1.
    if (offset > value.length - count) {
        throw new StringIndexOutOfBoundsException(offset + count);
    }
    this.value = Arrays.copyOfRange(value, offset, offset+count);
}
```

从上述代码可以看到，在新版本的 substring()中，不再复用原 String 的 value，而是将实际需要的部分做了复制，该问题也得到了完全的修复。

7.2.3 有关 String 常量池的位置

在虚拟机中，有一块称为常量池的区间专门用于存放字符串常量。在 JDK 1.6 之前，这块区间属于永久区的一部分，但是在 JDK 1.7 以后，它就被移到了堆中进行管理。

【示例 7-5】请看下面的例子。

```
public class StringInternOOM {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        int i = 0;
        while(true){
            list.add(String.valueOf(i++).intern());
        }
    }
}
```

上述代码使用 String.intern()方法获得在常量池中的字符串引用，如果常量池中没有该常量字符串，该方法会将字符串加入常量池。然后，将该引用放入 list 进行持有，确保不被回收。使用如下参数运行这段程序：

```
-Xmx5m -XX:MaxPermSize=5m
```

在 JDK 1.6 中抛出错误如下：

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
 at java.lang.String.intern(Native Method)
 at geym.zbase.ch7.string.StringInternOOM.main(StringInternOOM.java:16)
```

在 JDK 1.7 中抛出错误如下：

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
 at java.util.Arrays.copyOf(Arrays.java:2245)
 at java.util.Arrays.copyOf(Arrays.java:2219)
 at java.util.ArrayList.grow(ArrayList.java:242)
 at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:216)
 at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:208)
 at java.util.ArrayList.add(ArrayList.java:440)
 at geym.zbase.ch7.string.StringInternOOM.main(StringInternOOM.java:16)
```

溢出的区域已经不同，JDK 1.6 中发生在永久区，而 JDK 1.7 则发生在堆中。这间接表明了常量池的位置变化。

另外一点值得注意的是，虽然 String.intern()的返回值永远等于字符串常量。但这并不代表在系统的每时每刻，相同的字符串的 intern()返回都会是一样的（虽然在 95%以上的情况下，都是相同的）。因为存在这么一种可能：在一次 intern()调用之后，该字符串在某一个时刻被回收，之后，再进行一次 intern()调用，那么字面量相同的字符串重新被加入常量池，但是引用位置已经不同。

```
public class ConstantPool {
    public static void main(String[] args) {
        if(args.length==0)return;
        System.out.println(System.identityHashCode((args[0]+Integer.
toString(0))));
        System.out.println(System.identityHashCode((args[0]+Integer.
toString(0)).intern()));
        System.gc();
        System.out.println(System.identityHashCode((args[0]+Integer.
toString(0)).intern()));
    }
}
```

上述代码接收一个参数，用于构造字符串，构造的字符串都是在原有字符串后加上字符串“0”。一共输出 3 次字符串的 Hash 值：第一次为字符串本身，第二次为常量池引用，第三次为进行了常量池回收后的相同字符串的常量池引用。程序的一种可能输出如下：

```
3916375
22279806
3154093
```

可以看到，3 次 Hash 值都是不同的。但是如果不进行程序当中的显式 GC 操作，那么后两次 Hash 值理应是相同的，读者可以自行尝试。

7.3　虚拟机也有内窥镜：使用 MAT 分析 Java 堆

MAT 是 Memory Analyzer 的简称，它是一款功能强大的 Java 堆内存分析器。可以用于查找内存泄露以及查看内存消耗情况。MAT 是基于 Eclipse 开发的，是一款免费的性能分析工具。读者可以在 http://www.eclipse.org/mat/下载并使用 MAT。

7.3.1　初识 MAT

在分析堆快照前，首先需要导出应用程序的堆快照。在本书前文中提到的 jmap、JConsole 和 Visual VM 等工具都可用于获得 Java 应用程序的堆快照文件。此外，MAT 本身也具有这个功能。

如图 7.4 所示，单击“Accquire Heap Dump”菜单后，会弹出当前 Java 应用程序列表，选择要分析的应用程序即可，如图 7.5 所示。

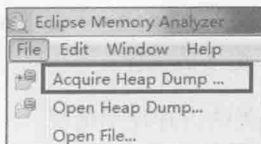


图 7.4　MAT 获取堆快照

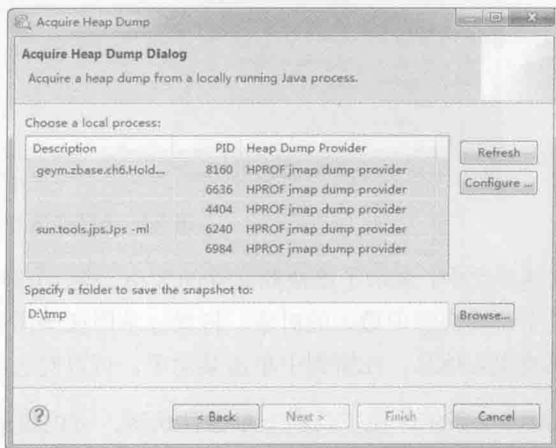


图 7.5　导出指定程序堆快照

除了直接在 MAT 中导出正在运行的应用程序堆快照外，也可以通过“Open Heap Dump”来打开一个既存的堆快照文件。

注意：使用 MAT 既可以打开一个已有的堆快照，也可以通过 MAT 直接从活动 Java 程序中导出堆快照。

如图 7.6 所示，显示了正常打开堆快照文件后的 MAT 的界面。

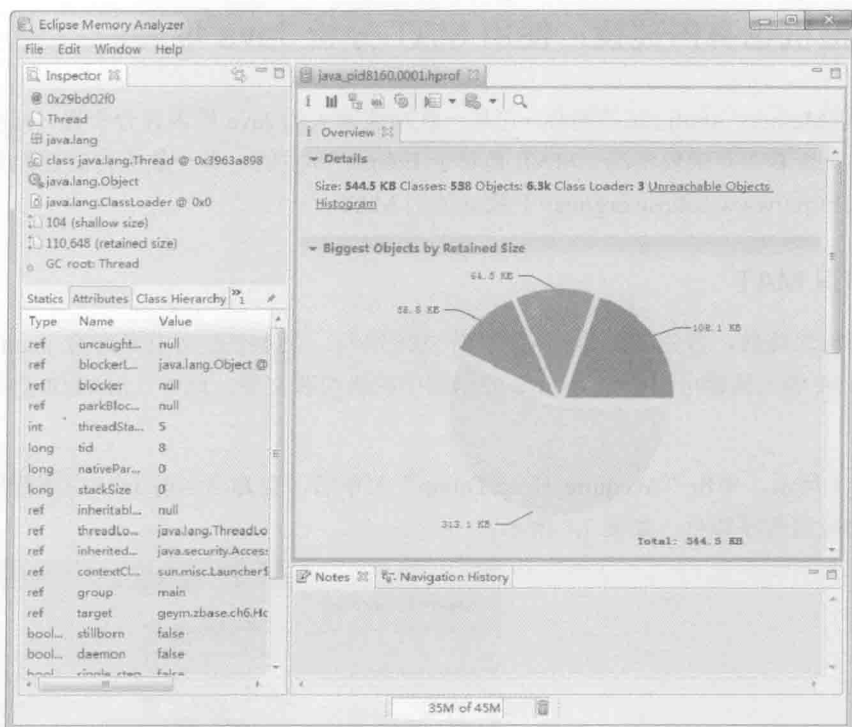


图 7.6 MAT 运行界面

右侧界面中，显示了堆快照文件的大小、类、实例和 ClassLoader 的总数。在右侧的饼图中，显示了当前堆快照中最大的对象。将鼠标悬停在饼图中，可以在左侧的 Inspector 界面中，查看该对象的相应信息。在饼图中单击某对象，可以对选中的对象进行更多的操作。

如图 7.7 所示，在工具栏上单击柱状图，可以显示系统中所有类的内存使用情况。



图 7.7 通过 MAT 工具栏查看内存使用情况

图 7.8 为系统内所有类的统计信息，包含类的实例数量和占用的空间。

另外一个实用的功能是，可以通过 MAT 查看系统中的 Java 线程，如图 7.9 所示。

当然，这里查看 Java 层面的应用线程，对于虚拟机的系统线程是无法显示的。通过线程的堆栈，还可以查看局部变量的信息。如图 7.10 所示，带有“<local>”标记的，就为当前帧栈的局部变量，这部分信息可能存在缺失。

java_pid8160.0001.hprof

Class Name	Objects	Shallow Heap
<Regex>	<Numeric>	<Numeric>
char[]	1,382	364,688
byte[]	14	57,952
java.lang.String	1,646	39,504
java.util.TreeMap$Entry	771	24,672
java.lang.Object[]	307	12,192

图 7.8　MAT 查看类的柱状图



图 7.9　通过 MAT 工具栏查看 Java 线程

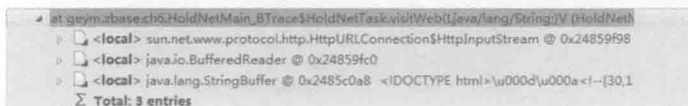


图 7.10　局部变量信息

MAT 的另外一个常用功能，是在各个对象的引用列表中穿梭查看。对于给定一个对象，通过 MAT 可以找到引用当前对象的对象，即入引用（Incomming References），以及当前对象引用的对象，即出引用（Outgoing References），如图 7.11 所示。

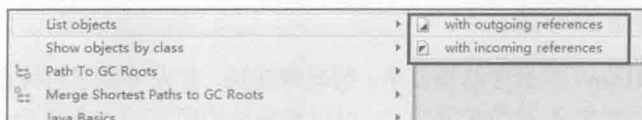


图 7.11　显示对象引用信息

图 7.12 显示了由 HttpURLConnection$HttpInputStream 对象开始的 outgoing 引用链，可以看到，顺着该对象查找，可以依次找到 HttpURLConnection 对象和 Java.net.URL 对象，这说明在 HttpURLConnection$HttpInputStream 对象内部引用了 HttpURLConnection，而在 HttpURLConnection 内部则引用了 Java.net.URL。



图 7.12　显示 outgoing 对象引用

7.3.2 浅堆和深堆

浅堆（Shallow Heap）和深堆（Retained Heap）是两个非常重要的概念，它们分别表示一个对象结构所占用的内存大小和一个对象被 GC 回收后，可以真实释放的内存大小。

浅堆（Shallow Heap）是指一个对象所消耗的内存。在 32 位系统中，一个对象引用会占据 4 个字节，一个 int 类型会占据 4 个字节，long 型变量会占据 8 个字节，每个对象头需要占用 8 个字节。

根据堆快照格式不同，对象的大小可能会向 8 字节进行对齐。以 String 对象为例，如图 7.13 所示，显示了 String 对象的几个属性（JDK 1.7，与 JDK 1.6 有差异）。

int	hash32	0
int	hash	0
ref	value	C:\Users\Administrat

图 7.13　JDK 1.7 中 String 结构

2 个 int 值共占 8 字节，对象引用占用 4 字节，对象头 8 字节，合计 20 字节，向 8 字节对齐，故占 24 字节。

这 24 字节为 String 对象的浅堆大小。它与 String 的 value 实际取值无关，无论字符串长度如何，浅堆大小始终是 24 字节。

深堆（Retained Heap）的概念略微复杂。要理解深堆，首先需要了解保留集（Retained Set）。对象 A 的保留集指当对象 A 被垃圾回收后，可以被释放的所有的对象集合（包括对象 A 本身），即对象 A 的保留集可以被认为是只能通过对象 A 被直接或间接访问到的所有对象的集合。通俗地说，就是指仅被对象 A 所持有的对象的集合。深堆是指对象的保留集中所有的对象的浅堆大小之和。

注意： 浅堆指对象本身占用的内存，不包括其内部引用对象的大小。一个对象的深堆指只能通过该对象访问到的（直接或间接）所有对象的浅堆之和，即对象被回收后，可以释放的真实空间。

另外一个常用的概念是对象的实际大小。这里，对象的实际大小定义为一个对象所能触及的所有对象的浅堆大小之和，也就是通常意义上我们说的对象大小。与深堆相比，似乎这个在日常开发中更为直观和被人接受，但实际上，这个概念和垃圾回收无关。

如图 7.14 所示，显示了一个简单的对象引用关系图，对象 A 引用了 C 和 D，对象 B 引用了 C 和 E。那么对象 A 的浅堆大小只是 A 本身，不含 C 和 D，而 A 的实际大小为 A、C、D 三者之和。而 A 的深堆大小为 A 与 D 之和，由于对象 C 还可以通过对象 B 访问到，因此不在对

象 A 的深堆范围内。

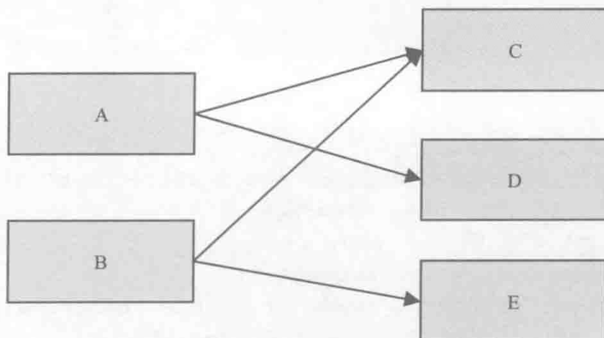


图 7.14　对象引用和深堆大小

7.3.3　例解 MAT 堆分析

在了解了浅堆、深堆和 MAT 的基本使用方法后，本节将通过一个简单的小案例，展示堆文件的分析方法。

【示例 7-6】在本案例中，设想这样一个场景：有一个学生浏览网页的记录程序，它将记录每个学生访问过的网站地址。它由三个部分组成：Student、WebPage 和 TraceStudent 三个类。它们的实现如下（本书使用 32 位 JDK 演示，64 位 JDK 的对象头大于 32 位 JDK，数据上存在出入，望读者留意）。

Student 类：

```
public class Student {
    private int id;
    private String name;
    private List<WebPage> history=new Vector<WebPage>();

    public Student(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
//省略 setter 和 getter 方法
}
```

WebPage 类：

```
public class WebPage {
    private String url;
```

```
    private String content;
//省略 getter 和 setter
}
```

TraceStudent 类：

```
public class TraceStudent {
    static List<WebPage> webpages = new Vector<WebPage>();
    public static void createWebPages() {
        for (int i = 0; i < 100; i++) {
            WebPage wp = new WebPage();
            wp.setUrl("http://www." + Integer.toString(i) + ".com");
            wp.setContent(Integer.toString(i));
            webpages.add(wp);
        }
    }
    public static void main(String[] args) {
        createWebPages();
        Student st3 = new Student(3, "billy");
        Student st5 = new Student(5, "alice");
        Student st7 = new Student(7, "taotao");
        for (int i = 0; i < webpages.size(); i++) {
            if (i % st3.getId() == 0)
                st3.visit(webpages.get(i));
            if (i % st5.getId() == 0)
                st5.visit(webpages.get(i));
            if (i % st7.getId() == 0)
                st7.visit(webpages.get(i));
        }
        webpages.clear();
        System.gc();
    }
}
```

可以看到，在 TraceStudent 类中，首先创建了 100 个网址，为阅读方便，这里的网址均以数字作为域名，分别为 0~99。之后，程序创建了 3 名学生：billy、alice 和 taotao。他们分别浏览了能被 3、5、7 整除的网页。在程序运行后，3 名学生的 history 中应该保护他们各自访问过的网页。现在，希望在程序退出前，得到系统的堆信息，并加以分析，查看每个学生实际访问的网页地址。

使用如下参数运行程序：

```
-XX:+HeapDumpBeforeFullGC -XX:HeapDumpPath=D:/stu.hprof
```

使用 MAT 打开产生的 stu.hrof 文件。在线程视图中可以通过主线程，找到 3 名学生的引用，如图 7.15 所示，为读者阅读方便，这里已经标出了每个实例的学生名。除了对象名称外，MAT 还给出了浅堆大小和深堆大小。可以看到，所有 Student 类的浅堆统一为 24 字节，和它们持有的内容无关，而深堆大小各不相同，这和每名学生访问的网页有关。

Object / Stack Frame	Name	Shall...	Retain...
<Regex>	<Rege...	<Num...	<Num...
java.lang.Thread @ 0x24680a90	main	104	8,512
at java.lang.Runtime.gc()V (Native Method)			
at java.lang.System.gc()V (System.java:983)			
at geym.zbase.ch7.heap.TraceStudent.main([Ljava/lang/String;)V (Tr			
<local> java.lang.String[0] @ 0x24704118		16	16
<local> geym.zbase.ch7.heap.Student @ 0x2470dd50	billy	24	3,216
<local> geym.zbase.ch7.heap.Student @ 0x2470dde8	alice	24	1,600
<local> geym.zbase.ch7.heap.Student @ 0x2470de80	taotao	24	1,216
Σ Total: 4 entries			
Σ Total: 3 entries			

图 7.15　在堆中显示 3 名学生

为了获得 taotao 同学访问过的网页，可以在 taotao 的记录中通过“出引用”（Outgoing References）查找，就可以找到由 taotao 可以触及的对象，也就是他访问过的网页，如图 7.16 所示。

<Regex>	<Numeric>	<Numeric>
geym.zbase.ch7.heap.Student @ 0x2470de80	24	1,216
<class> class geym.zbase.ch7.heap.Student @ 0x2470dce0	0	0
name java.lang.String @ 0x2470de98 taotao	24	48
history java.util.Vector @ 0x2470dec8	24	1,144
<class> class java.util.Vector @ 0x3963c798 System Class	16	16
elementData java.lang.Object[20] @ 0x2470e088	96	1,120
<class> class java.lang.Object[] @ 0x3963c150	0	0
[0] geym.zbase.ch7.heap.WebPage @ 0x247055e8	16	128
<class> class geym.zbase.ch7.heap.WebPage @ 0x247	0	0
url java.lang.String @ 0x24705a68 http://www.0.com	24	72
content java.lang.String @ 0x24705ac0 0	24	40
Σ Total: 3 entries		
[1] geym.zbase.ch7.heap.WebPage @ 0x247060d8	16	128
<class> class geym.zbase.ch7.heap.WebPage @ 0x247	0	0
url java.lang.String @ 0x24706168 http://www.7.com	24	72
content java.lang.String @ 0x247061c0 7	24	40
Σ Total: 3 entries		
[2] geym.zbase.ch7.heap.WebPage @ 0x24706838	16	128
[3] geym.zbase.ch7.heap.WebPage @ 0x24706fe8	16	128
[4] geym.zbase.ch7.heap.WebPage @ 0x247076e8	16	128
[5] geym.zbase.ch7.heap.WebPage @ 0x24707de8	16	128
[6] geym.zbase.ch7.heap.WebPage @ 0x24708638	16	128

图 7.16　查找 taotao 访问过的网址

可以看到，堆中完整显示了所有 taotao 同学的 history 中的网址页面（都是可以被 7 整除的网址）。

如果现在希望查看哪些同学访问了“http://www.0.com”，则可以在对应的 WebPage 对象中通过“入引用”（Incoming References）查找。如图 7.17 所示，显然这个网址被 3 名学生都访问过了。

<Regex>	<Numeric>	<Numeric>
geym.zbase.ch7.heap.WebPage @ 0x247055e8	16	128
[0] java.lang.Object[20] @ 0x2470df78	96	1,504
elementData java.util.Vector @ 0x2470de30	24	1,528
history geym.zbase.ch7.heap.Student @ 0x2470dde8	24	1,600
[0] java.lang.Object[40] @ 0x2470dfd8	176	3,120
elementData java.util.Vector @ 0x2470dd98	24	3,144
history geym.zbase.ch7.heap.Student @ 0x2470dd50	24	3,216
[0] java.lang.Object[20] @ 0x2470e088	96	1,120
elementData java.util.Vector @ 0x2470dec8	24	1,144
history geym.zbase.ch7.heap.Student @ 0x2470de80	24	1,216
Σ **Total: 3 entries**		

图 7.17　通过入引用查找浏览过 www.0.com 的学生

下面，在这个实例中，再来理解一下深堆的概念，如图 7.18 所示，在 taotao 同学的访问历史中，一共有 15 条数据，每一条 WebPage 占用 128 字节的空间（深堆），而 15 条数据合计共占用 1920 字节。而 history 中的 elementData 数组实际深堆大小为 1120 字节。这是因为部分网址 WebPage 既被 taotao 访问，又被其他学生访问，因此 taotao 并不是唯一可以引用到它们的对象，对于这些对象的大小，自然不应该算在 taotao 同学的深堆中。根据程序的规律，只要被 3 或者 5 整除的网址，都不应该计算在内，满足条件的网址（能被 3 和 7 整除，或者能被 5 和 7 整除）有 0、21、35、42、63、70、84 等 7 个。它们合计大小为 7*128= 896 字节，故 taotao 的 history 对象中的 elementData 数组的深堆大小为 1920-896+96= 1120 字节。这里的 96 字节表示 elementData 数组的浅堆大小，由于 elementData 数组长度为 20（第 15~19 项为 null），每个引用 4 字节，合计 4*20=80 字节，数组对象头 8 字节，数组长度占 4 字节，合计 80+8+4=92 字节，向 8 字节对齐填充后，为 96 字节。

<class> class geym.zbase.ch7.heap.Student @ 0x2470dce0	0	0
name java.lang.String @ 0x2470de98 taotao	24	48
history java.util.Vector @ 0x2470dec8	24	1,144
<class> class java.util.Vector @ 0x3963c798 System Class	16	16
elementData java.lang.Object[20] @ 0x2470e088	96	1,120
<class> class java.lang.Object[] @ 0x3963c150	0	0
[0] geym.zbase.ch7.heap.WebPage @ 0x247055e8	16	128
[1] geym.zbase.ch7.heap.WebPage @ 0x247060d8	16	128
[2] geym.zbase.ch7.heap.WebPage @ 0x24706838	16	128
[3] geym.zbase.ch7.heap.WebPage @ 0x24706fe8	16	128
[4] geym.zbase.ch7.heap.WebPage @ 0x247076e8	16	128
[5] geym.zbase.ch7.heap.WebPage @ 0x24707de8	16	128
[6] geym.zbase.ch7.heap.WebPage @ 0x24708638	16	128
[7] geym.zbase.ch7.heap.WebPage @ 0x24708d38	16	128
[8] geym.zbase.ch7.heap.WebPage @ 0x24709438	16	128
[9] geym.zbase.ch7.heap.WebPage @ 0x24709b38	16	128
[10] geym.zbase.ch7.heap.WebPage @ 0x2470a238	16	128
[11] geym.zbase.ch7.heap.WebPage @ 0x2470a938	16	128
[12] geym.zbase.ch7.heap.WebPage @ 0x2470b2c8	16	128
[13] geym.zbase.ch7.heap.WebPage @ 0x2470b9c8	16	128
[14] geym.zbase.ch7.heap.WebPage @ 0x2470c0c8	16	128
Σ **Total: 16 entries**		

图 7.18　对象数据的深堆大小

7.3.4　支配树（Dominator Tree）

MAT 提供了一个称为支配树（Dominator Tree）的对象图。支配树体现了对象实例间的支配关系。在对象引用图中，所有指向对象 B 的路径都经过对象 A，则认为对象 A 支配对象 B。如果对象 A 是离对象 B 最近的一个支配对象，则认为对象 A 为对象 B 的直接支配者。支配树是基于对象间的引用图所建立的，它有以下基本性质：

- 对象 A 的子树（所有被对象 A 支配的对象集合）表示对象 A 的保留集（retained set），即深堆。
- 如果对象 A 支配对象 B，那么对象 A 的直接支配者也支配对象 B。
- 支配树的边与对象引用图的边不直接对应。

如图 7.19 所示，左图表示对象引用图，右图表示左图所对应的支配树。对象 A 和 B 由根对象直接支配，由于在到对象 C 的路径中，可以经过 A，也可以经过 B，因此对象 C 的直接支配者也是根对象。对象 F 与对象 D 相互引用，因为到对象 F 的所有路径必然经过对象 D，因此，对象 D 是对象 F 的直接支配者。而到对象 D 的所有路径中，必然经过对象 C，即使是从对象 F 到对象 D 的引用，从根节点出发，也是经过对象 C 的，所以，对象 D 的直接支配者为对象 C。

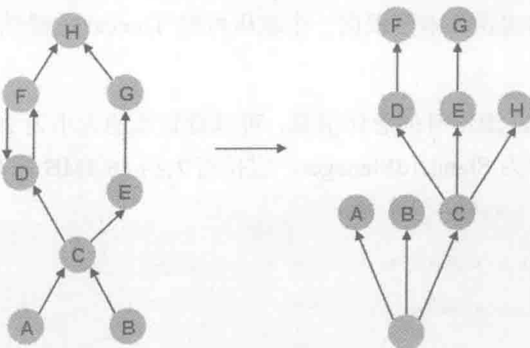


图 7.19　引用关系与支配树

同理，对象 E 支配对象 G。到达对象 H 的可以通过对象 D，也可以通过对象 E，因此对象 D 和 E 都不能支配对象 H，而经过对象 C 既可以到达 D 也可以到达 E，因此对象 C 为对象 H 的直接支配者。

在 MAT 中，单击工具栏上的对象支配树按钮，可以打开对象支配树视图，如图 7.20 所示。



图 7.20　从工具栏打开支配树

图 7.21 显示了对象支配树视图的一部分。该截图显示部分 billy 学生的 history 队列的直接支配对象。即当 billy 对象被回收，也会一并回收的所有对象。显然能被 5 或者 7 整除的网页不会出现在该列表中，因为它们同时被另外两名学生对象引用。

geym.zbase.ch7.heap.Student @ 0x2470dd50	24	3,216	0.84%
java.util.Vector @ 0x2470dd98	24	3,144	0.82%
java.lang.Object[40] @ 0x2470dfd8	176	3,120	0.81%
geym.zbase.ch7.heap.WebPage @ 0x24705cd8	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x24705fd8	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x247062d8	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x24706638	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x24706c38	16	128	0.03%
geym.zbase.ch7.heap.WebPage @ 0x247072e8	16	128	0.03%

图 7.21　支配树显示结果

注意：对象支配树中，某一个对象的子树，表示在该对象被回收后，也将被回收的对象的集合。

7.3.5　Tomcat 堆溢出分析

Tomcat 是最常用的 Java Servlet 容器之一，同时也可以当做单独的 Web 服务器使用。Tomcat 本身使用 Java 实现，并运行于 Java 虚拟机之上。在大规模请求时，Tomcat 有可能会因为无法承受压力而发生内存溢出错误。本节根据一个被压垮的 Tomcat 的堆快照文件，来分析 Tomcat 在崩溃时的内部情况。

图 7.22 显示了 Tomcat 溢出时的总体信息，可以看到堆的大小为 29.7MB。从统计饼图中得知，当前深堆最大的对象为 StandardManager，它持有大约 16.4MB 的对象。

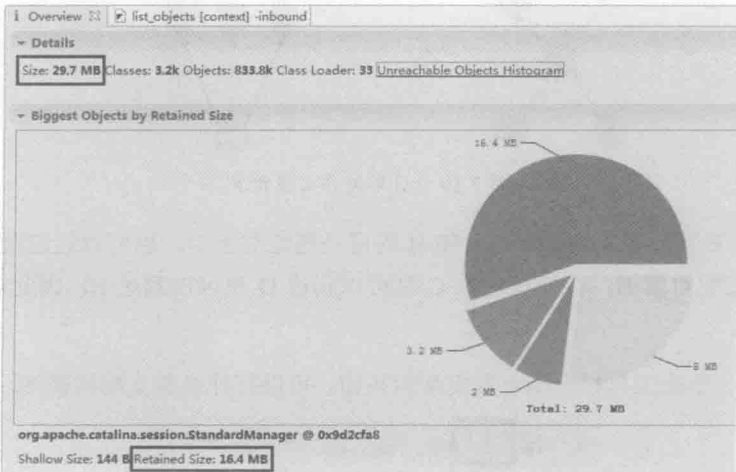


图 7.22　Tomcat 堆溢出总体信息

一般来说，我们总是会对占用空间最大的对象特别感兴趣，如果可以查看 StandardManager 内部究竟引用了哪些对象，对于分析问题可能会起到很大的帮助。因此，在饼图中单击 StandardManager 所在区域，在弹出菜单中选择"with outgoing references"命令，如图 7.23 所示。这样将会列出被 StandardManager 引用的所有对象。

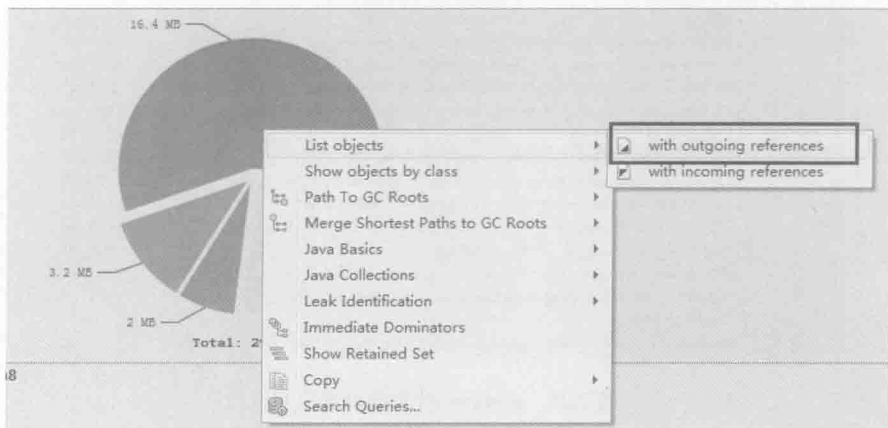


图 7.23　显示 StandardManager 的引用对象

图 7.24 显示了被 StandardManager 引用的对象，其中特别显眼的就是 sessions 对象，它占用了约 17MB 空间。

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
org.apache.catalina.session.StandardManager @ 0x9d2cfa8	144	17,211,600
sessions java.util.concurrent.ConcurrentHashMap @ 0x9d2d0f0	40	17,201,792
container org.apache.catalina.core.StandardContext @ 0x98023b0	520	311,592
mserver com.sun.jmx.mbeanserver.JmxMBeanServer @ 0x95cc6e8	32	29,832
sessionCreationTiming java.util.LinkedList @ 0x9d2d080	24	4,848
sessionExpirationTiming java.util.LinkedList @ 0x9d2d0b0	24	2,448

图 7.24　被 StandardManager 引用的 sessions

继续查找，打开 sessions 对象，查看被它引用的对象，如图 7.25 所示。可以看到 sessions 对象为 ConcurrentHashMap，其内部分为 16 个 Segment。从深堆大小看，每个 Segment 都比较平均，大约为 1MB，合计 17MB。

继续打开 Segment，查看存储在 sessions 中的真实对象。如图 7.26 所示，可以找到内部存放的为 StandardSession 对象。

<Regex>	<Numeric>	<Numeric>
org.apache.catalina.session.StandardManager @ 0x9d2cfa8	144	17,211,600
sessions java.util.concurrent.ConcurrentHashMap @ 0x9d2d0f0	40	17,201,792
segments java.util.concurrent.ConcurrentHashMap$Segment[16] @	80	17,201,752
[3] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,151,440
[4] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,117,008
[11] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2	32	1,115,320
[8] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,110,160
[15] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2	32	1,099,808
[0] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,094,672
[7] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,087,800
[5] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,086,048
[9] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,080,920
[10] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2	32	1,079,200
[6] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,074,040
[2] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,070,600
[14] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2	32	1,046,488
[1] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d	32	1,017,248
[12] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2	32	989,760
[13] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2	32	981,160
<class> class java.util.concurrent.ConcurrentHashMap$Segment	0	0
Σ **Total: 17 entries**		
<class> class java.util.concurrent.ConcurrentHashMap @ 0x5756a4(	32	32

图 7.25　sessions 对象的内部引用

sessions java.util.concurrent.ConcurrentHashMap @ 0x9d2d0f0	40	17,201,792
segments java.util.concurrent.ConcurrentHashMap$Segment[16] @ 0x9d2d118	80	17,201,752
[3] java.util.concurrent.ConcurrentHashMap$Segment @ 0x9d2d408	32	1,151,440
table java.util.concurrent.ConcurrentHashMap$HashEntry[1024] @ 0xaaddbc8	4,112	1,151,352
[112] java.util.concurrent.ConcurrentHashMap$HashEntry @ 0xaad9318	24	8,600
next java.util.concurrent.ConcurrentHashMap$HashEntry @ 0xaa6d370	24	6,880
value org.apache.catalina.session.StandardSession @ 0xaad8ca0	80	1,592
key java.lang.String @ 0xaad92b0 D54FB440CBF6A221493DD7AF99767	24	104
<class> class java.util.concurrent.ConcurrentHashMap$HashEntry @ 0x!	0	0
Σ **Total: 4 entries**		
[536] java.util.concurrent.ConcurrentHashMap$HashEntry @ 0xab23a28	24	6,880
next java.util.concurrent.ConcurrentHashMap$HashEntry @ 0xaa3dda8	24	5,160
value org.apache.catalina.session.StandardSession @ 0xab233b0	80	1,592
key java.lang.String @ 0xab239c0 019B9DA95527925E61DDE9DF9AA5!	24	104
<class> class java.util.concurrent.ConcurrentHashMap$HashEntry @ 0x!	0	0
Σ **Total: 4 entries**		
[953] java.util.concurrent.ConcurrentHashMap$HashEntry @ 0xaf86958	24	6,880
next java.util.concurrent.ConcurrentHashMap$HashEntry @ 0xadfced0	24	5,160
value org.apache.catalina.session.StandardSession @ 0xaf862e0	80	1,592
key java.lang.String @ 0xaf868f0 2520207DD1BAD988FA48BCFD207F2E	24	104
<class> class java.util.concurrent.ConcurrentHashMap$HashEntry @ 0x!	0	0
Σ **Total: 4 entries**		

图 7.26　通过 MAT 找到 StandardSession 对象

通过 OQL 命令，查找所有的 StardardSession，如图 7.27 所示（有关 OQL，请参阅 7.4 节）。可以看到当前堆中含有 9941 个 session，并且每一个 session 的深堆为 1592 字节，合计约 15MB，达到当前堆大小的 50%。由此，可以知道，当前 Tomcat 发生内存溢出的原因，极可能是由于在短期内接收大量不同客户端的请求，从而创建大量 session 导致。

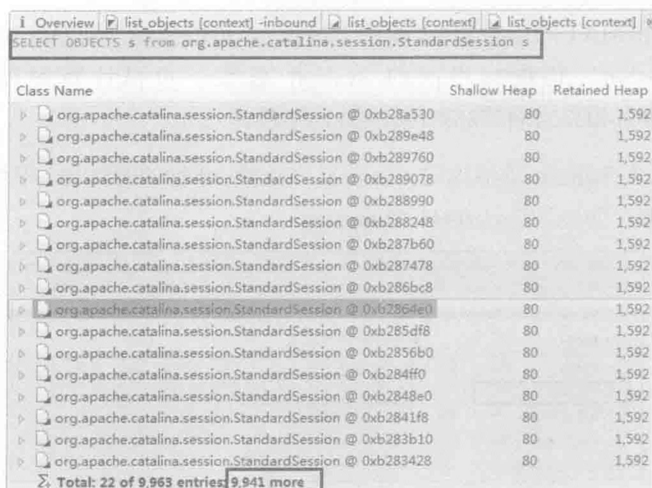


图 7.27　通过 OQL 查找所有的 session 对象

为了获得更为精确的信息，可以查看每一个 session 的内部数据，如图 7.28 所示，在左侧的对象属性表中，可以看到所选中的 session 的最后访问时间和创建时间。

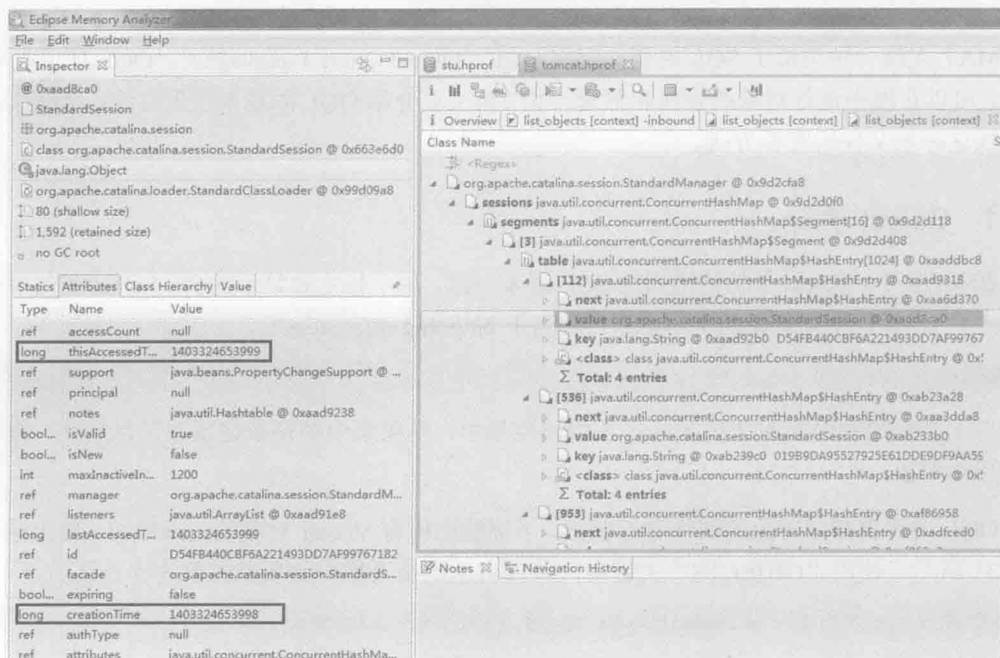


图 7.28　session 的内部数据

通过 OQL 命令和 MAT 的排序功能，如图 7.29 所示，可以找到当前系统中最早创建的 session 和最后创建的 session。再根据当前的 session 总数，可以计算每秒的平均压力为：9941/(1403324677648-1403324645728)*1000=311 次/秒。

由此推断，在发生 Tomcat 堆溢出时，Tomcat 在连续 30 秒的时间内，平均每秒接收了约 311 次不同客户端的请求，创建了合计 9941 个 session。

i Overview | list_objects [context] -inbound | list_objects [context] | list_objects

SELECT s.creationTime FROM org.apache.catalina.session.StandardSession s

s.creationTime	s.creationTime
<Numeric>	<Numeric>
1,403,324,645,728	1,403,324,677,648
1,403,324,645,755	1,403,324,677,129
1,403,324,645,759	1,403,324,676,355
1,403,324,645,763	

图 7.29　查找最早和最晚创建的 session

7.4　筛选堆对象：MAT 对 OQL 的支持

MAT 支持一种类似于 SQL 的查询语言 OQL（Object Query Language）。OQL 使用类 SQL 语法，可以在堆中进行对象的查找和筛选。本节将主要介绍 OQL 的基本使用方法，帮助读者尽快掌握这种堆文件的查看方式。

7.4.1　Select 子句

在 MAT 中，Select 子句的格式与 SQL 基本一致，用于指定要显示的列。Select 子句中可以使用“*”，查看结果对象的引用实例（相当于 outgoing references）。

```
SELECT * FROM java.util.Vector v
```

以上查询的输出如图 7.30 所示，在输出结果中，结果集中的每条记录都可以展开，查看各自的引用对象。

OQL 还可以指定对象的属性进行输出，下例输出所有 Vector 对象的内部数组，输出结果如图 7.31 所示。使用“OBJECTS”关键字，可以将返回结果集中的项以对象的形式显示。

```
SELECT OBJECTS v.elementData FROM java.util.Vector v
```

i Overview | OQL

SELECT * FROM java.util.Vector v

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Vector @ 0x2470dec8	24	1,144
<class> class java.util.Vector @ 0x3963	16	16
elementData java.lang.Object[20] @ 0x	96	1,120
Σ Total: 2 entries		
java.util.Vector @ 0x2470de30	24	1,528
java.util.Vector @ 0x2470dd98	24	3,144
java.util.Vector @ 0x247040c8	24	680
java.util.Vector @ 0x246f95e8	24	80
java.util.Vector @ 0x246f9508	24	768
java.util.Vector @ 0x246f6768	24	80
java.util.Vector @ 0x246f63d8	24	80
java.util.Vector @ 0x24680ce0	24	80
java.util.Vector @ 0x24680c90	24	80
Σ Total: 10 entries		

图 7.30　Select 查询返回结构

i Overview | OQL

SELECT objects v.elementData FROM java.util.Vector v

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.Object[20] @ 0x2470e088	96	1,120
java.lang.Object[20] @ 0x2470df78	96	1,504
java.lang.Object[40] @ 0x2470dfd8	176	3,120
java.lang.Object[160] @ 0x2470ad38	656	656
java.lang.Object[10] @ 0x246f9600	56	56
java.lang.Object[10] @ 0x246f9520	56	744
java.lang.Object[10] @ 0x246f6780	56	56
java.lang.Object[10] @ 0x246f63f0	56	56
java.lang.Object[10] @ 0x24680cf8	56	56
java.lang.Object[10] @ 0x24680ca8	56	56
Σ Total: 10 entries		

图 7.31　指定查询属性

下例显示 String 对象的 char 数组（用于 JDK 1.7 的堆）：

```
SELECT OBJECTS s.value FROM java.lang.String s
```

在 Select 子句中，使用“AS RETAINED SET”关键字可以得到所得对象的保留集。下例得到 geym.zbase.ch7.heap.Student 对象的保留集，其结果如图 7.32 所示。

```
SELECT AS RETAINED SET * FROM geym.zbase.ch7.heap.Student
```

i Overview | OQL | OQL

SELECT AS RETAINED SET * FROM geym.zbase.ch7.heap.Student

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
geym.zbase.ch7.heap.WebPage @ 0x247055e8	16	128
java.lang.String @ 0x24705a68 http://www.0.com	24	72
char[16] @ 0x24705a80 http://www.0.com	48	48
char[1] @ 0x24705ab0 0	16	16
java.lang.String @ 0x24705ac0 0	24	40
geym.zbase.ch7.heap.WebPage @ 0x24705cd8	16	128
java.lang.String @ 0x24705d68 http://www.3.com	24	72
char[16] @ 0x24705d80 http://www.3.com	48	48
char[1] @ 0x24705db0 3	16	16
java.lang.String @ 0x24705dc0 3	24	40
geym.zbase.ch7.heap.WebPage @ 0x24705ed8	16	128
java.lang.String @ 0x24705f68 http://www.5.com	24	72
char[16] @ 0x24705f80 http://www.5.com	48	48
char[1] @ 0x24705fb0 5	16	16
java.lang.String @ 0x24705fc0 5	24	40
geym.zbase.ch7.heap.WebPage @ 0x24705fd8	16	128
java.lang.String @ 0x24706068 http://www.6.com	24	72
char[16] @ 0x24706080 http://www.6.com	48	48
char[1] @ 0x247060b0 6	16	16

图 7.32　查询对象保留集

“DISTINCT”关键字用于在结果集中去除重复对象。下例的输出如图 7.33 所示，输出结果中只有一条“class java.lang.String”记录。如果没有“DISTINCT”，那么查询将为每个 String 实例输出其对应的 Class 信息。

```
SELECT DISTINCT OBJECTS classof(s) FROM java.lang.String s
```

i Overview | OQL | OQL

SELECT DISTINCT OBJECTS classof(s) FROM java.lang.String s

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
class java.lang.String @ 0x39639950 **System Class, Native**	24	48
<class> class java.lang.Class @ 0x39639b28 System Cl	40	56
<classloader> java.lang.ClassLoader @ 0x0 <system c	56	56
serialPersistentFields java.io.ObjectStreamField[0] @ 0	16	16
CASE_INSENSITIVE_ORDER java.lang.String$CaseInsens	8	8
<super> class java.lang.Object @ 0x39639790 System	0	0
Σ **Total: 5 entries**		

图 7.33　DISTINCT 关键字的使用

7.4.2　From 子句

From 子句用于指定查询范围，它可以指定类名、正则表达式或者对象地址。

下例使用 From 子句，指定类名进行搜索，并输出所有的 java.lang.String 实例。

```
SELECT * FROM java.lang.String s
```

下例使用正则表达式，限定搜索范围，输出所有 geym.zbase 包下所有类的实例，如图 7.34 所示。

```
SELECT * FROM "geym\.zbase\..*"
```

i Overview | OQL | OQL

SELECT * FROM "geym\.zbase\..*"

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
geym.zbase.ch7.heap.WebPage @ 0x2470c1c8	16	128
geym.zbase.ch7.heap.WebPage @ 0x2470c0c8	16	128
geym.zbase.ch7.heap.WebPage @ 0x2470bec8	16	128
geym.zbase.ch7.heap.WebPage @ 0x2470bdc8	16	128
geym.zbase.ch7.heap.WebPage @ 0x2470bbc8	16	128
geym.zbase.ch7.heap.WebPage @ 0x2470b9c8	16	128

图 7.34　正则表达式查询

也可以直接使用类的地址进行搜索。使用类的地址的好处是可以区分被不同 ClassLoader 加载的同一种类型。下例中“0x37a014d8”为类的地址。

```
select * from 0x37a014d8
```

有多种方法可以获得类的地址，在 MAT 中，一种最为简单的方法如图 7.35 所示。

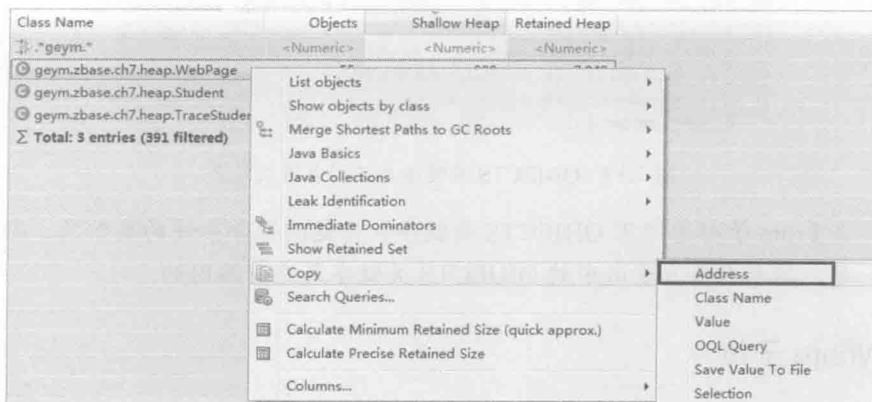


图 7.35　复制对象地址

在 From 子句中，还可以使用“INSTANCEOF”关键字，返回指定类的所有子类实例。下例的查询返回了当前堆快照中所有的抽象集合实例，包括 java.util.Vector、java.util.ArrayList 和 java.util.HashSet 等。

```
SELECT * FROM INSTANCEOF java.util.AbstractCollection
```

在 From 子句中，还可以使用“OBJECTS”关键字。使用“OBJECTS”关键字后，那么原本应该返回类的实例的查询，将返回类的信息。

```
SELECT * FROM OBJECTS java.lang.String
```

以上查询的返回结果如图 7.36 所示。它仅返回一条记录，表示 java.lang.String 的类的信息。如果不使用“OBJECTS”关键字，这个查询将返回所有的 java.lang.String 实例。

i Overview　OQL　OQL　Histogram

SELECT * FROM OBJECTS java.lang.String

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
class java.lang.String @ 0x39639950 **System Class, Native Stack**	24	48

图 7.36　OBJECTS 关键字用于 FROM 子句

“OBJECTS”关键字也支持与正则表达式一起使用。下面的查询，返回了所有满足给定正则表达式的所有类，其结果如图 7.37 所示。

```
SELECT * FROM OBJECTS "geym\.zbase\..*"
```

i Overview | OQL | OQL | Histogram

SELECT * FROM OBJECTS "geym\.zbase\..*"

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
class geym.zbase.ch7.heap.WebPage @ 0x24705578	0	0
class geym.zbase.ch7.heap.TraceStudent @ 0x247037f0	8	688
class geym.zbase.ch7.heap.Student @ 0x2470dce0	0	0
Σ Total: 3 entries		

图 7.37　OBJECTS 关键字与正则表达式结合

注意：在 From 子句中使用 OBJECTS 关键字，将返回符合条件的类信息，而非实例信息。这与 Select 子句中的 OBJECTS 关键字是完全不同的。

7.4.3　Where 子句

Where 子句用于指定 OQL 的查询条件。OQL 查询将只返回满足 Where 子句指定条件的对象。Where 子句的格式与传统 SQL 极为相似。

下例返回长度大于 10 的 char 数组。

```
SELECT * FROM char[] s WHERE s.@length>10
```

下例返回包含“java”子字符串的所有字符串，使用“LIKE”操作符，“LIKE”操作符的操作参数为正则表达式。

```
SELECT * FROM java.lang.String s WHERE toString(s) LIKE ".*java.*"
```

下例返回所有 value 域不为 null 的字符串，使用“=”操作符。

```
SELECT * FROM java.lang.String s where s.value!=null
```

Where 子句支持多个条件的 AND、OR 运算。下例返回数组长度大于 15，并且深堆大于 1000 字节的所有 Vector 对象。

```
SELECT * FROM java.util.Vector v WHERE v.elementData.@length>15 AND
v.@retainedHeapSize>1000
```

7.4.4　内置对象与方法

OQL 中可以访问堆内对象的属性，也可以访问堆内代理对象的属性。访问堆内对象的属性时，格式如下：

```
[ <alias>. ] <field> . <field>. <field>
```

其中 alias 为对象名称。

下例访问 java.io.File 对象的 path 属性，并进一步访问 path 的 value 属性。

```
SELECT toString(f.path.value) FROM java.io.File f
```

以上查询得到的结果如图 7.38 所示。

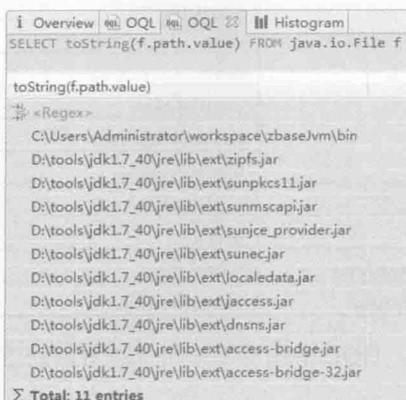


图 7.38　显示文件信息

这些堆内对象的属性与 Java 对象一致，拥有与 Java 对象相同的结果。

MAT 为了能快捷地获取堆内对象的额外属性（比如对象占用的堆大小、对象地址等），为每种元类型的堆内对象建立了相对应的代理对象，以增强原有的对象功能。访问代理对象的属性时，使用如下格式：

```
[ <alias>. ] @<attribute>
```

其中，alias 为对象名称，attribute 为属性名。

下例显示了 String 对象的内容、objectid 和 objectAddress。

```
SELECT s.toString(), s.@objectId, s.@objectAddress FROM java.lang.String s
```

下例显示了 File 对象的对象 ID、对象地址、代理对象的类型、类的类型、对象的浅堆大小以及对象的显示名称。

```
SELECT f.@objectId, f.@objectAddress, f.@class, f.@clazz, f.@usedHeapSize,
f.@displayName FROM java.io.File f
```

下例显示 java.util.Vector 内部数组的长度。

```
SELECT v.elementData.@length FROM java.util.Vector v
```

表 7.1 整理了 MAT 代理对象的基本属性。

表 7.1　MAT代理对象的基本属性

对象说明	对象名	对象方法/字段	对象方法/字段说明
基对象	IObejct	objectId	对象ID
		objectAddress	对象地址
		class	代理对象类型
		clazz	对象类类型
		usedHeapSize	浅堆大小
		retainedHeapSize	深堆大小
		displayName	显示名称
Class对象	IClass	classLoaderId	ClassLoad的ID
数组	IArray	length	数组长度
元类型数组	IPrimitiveArray	valueArray	数组内容
对象数组	IObjectArray	referenceArray	数组内容

除了使用代理对象的属性，OQL 中还可以使用代理对象的方法，使用格式如下：

```
[ <alias> . ] @<method>( [ <expression>, <expression> ] )
```

下例显示 int 数组中索引下标为 2 的数据内容。

```
SELECT s.getValueAt(2) FROM int[] s WHERE (s.@length > 2)
```

下例显示对象数组中索引下标为 2 的对象。

```
SELECT OBJECTS s.@referenceArray.get(2) FROM java.lang.Object[] s WHERE
(s.@length > 2)
```

下例显示了当前堆中所有的类型。

```
select * from ${snapshot}.getClasses()
```

下例显示了所有的 java.util.Vector 对象及其子类型，它的输出如图 7.39 所示。

```
select * from INSTANCEOF java.util.Vector
```

下例显示当前对象是否是数组。

```
SELECT c, classof(c).isArrayType() FROM ${snapshot}.getClasses() c
```

代理对象的方法整理如表 7.2 所示。

i Overview | default_report org.eclipse.mat.api:suspects | OQL

```
select * from INSTANCEOF java.util.Vector
```

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.util.Vector @ 0x2470dec8	24	1,144
java.util.Vector @ 0x2470de30	24	1,528
java.util.Vector @ 0x2470dd98	24	3,144
java.util.Vector @ 0x247040c8	24	680
java.util.Vector @ 0x246f95e8	24	80
java.util.Vector @ 0x246f9508	24	768
java.util.Vector @ 0x246f6768	24	80
java.util.Vector @ 0x246f63d8	24	80
java.util.Vector @ 0x24680ce0	24	80
java.util.Vector @ 0x24680c90	24	80
java.util.Stack @ 0x246f9940	24	80
java.util.Stack @ 0x246f6af8	24	80
java.util.Stack @ 0x24680d30	24	80
Σ Total: 13 entries		

图 7.39　getClassesByName()函数使用

表 7.2　MAT代理对象的方法

对象说明	对象名	对象方法	对象方法说明
全局快照	ISnapshot	getClasses()	所有实例的集合
		getClassesByName(String name, boolean includeSubClasses)	根据名称选取符合条件的实例
类对象	IClass	hasSuperClass()	是否有超类
		isArrayType()	是否是数组
基对象	IObject	getObjectAddress()	取得对象地址
元类型数组	IPrimitiveArray	getValueAt(int index)	取得数组中给定索引的数据
元类型数组，对象数组	[] or List	get(int index)	取得数组中给定索引的数据

MAT 的 OQL 中还内置一些有用的函数，如表 7.3 所示。

表 7.3　OQL中的内置函数

函数	说明
toHex(number)	转为16进制
toString(object)	转为字符串
dominators(object)	取得直接支配对象
outbounds(object)	取得给定对象引用的对象
inbounds(object)	取得引用给定对象的对象

续表

函数	说明
classof(object)	取得当前对象的类
dominatorof(object)	取得给定对象的直接支配者

下例显示所有长度为 15 的字符串内容（JDK 1.7 导出的堆）。

```
SELECT toString(s) FROM java.lang.String s WHERE ((s.value.@length = 15) and (s.value != null))
```

下例显示所有 geym.zbase.ch7.heap.Student 对象的直接支配对象。即给定对象回收后，将释放的对象集合。

```
SELECT objects dominators(s) FROM geym.zbase.ch7.heap.Student s
```

以上查询的输出如图 7.40 所示，显示 Student 对象支配了 3 个字符串和 3 个 Vector 对象。

i Overview | default_report org.eclipse.mat.api:suspects | OQL

SELECT objects dominators(s) FROM geym.zbase.ch7.heap.Student s

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.String @ 0x2470dd68 billy	24	48
java.lang.String @ 0x2470de00 alice	24	48
java.lang.String @ 0x2470de98 taotao	24	48
java.util.Vector @ 0x2470dd98	24	3,144
java.util.Vector @ 0x2470de30	24	1,528
java.util.Vector @ 0x2470dec8	24	1,144
Σ Total: 6 entries		

图 7.40　dominators()函数输出

函数 dominatorof()与 dominators()的功能相反，它获取直接支配当前对象的对象。

```
SELECT distinct objects dominatorof(s) FROM  geym.zbase.ch7.heap.Student s
```

以上查询的输出如图 7.41 所示，显示所有的 Student 对象直接被主线程支配。

i Overview | default_report org.eclipse.mat.api:suspects | OQL

SELECT distinct objects dominatorof(s) FROM geym.zbase.ch7.heap.Student

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
java.lang.Thread @ 0x24680a90 main Thread	104	8,512

图 7.41　dominatorof()函数输出

注意： 函数 dominatorof()与 dominators()的功能正好相反。dominatorof()用于获得直接支配当前对象的对象，而 dominators()用于获取直接支配对象。

下例取得引用 WebPage 的对象。

```
SELECT objects inbounds(w) FROM geym.zbase.ch7.heap.WebPage w
```

下例取得堆快照中所有在 geym.zbase 包中的存在对象实例的类型，其输出如图 7.42 所示。

```
SELECT distinct objects classof(obj) FROM "geym\.zbase\..*" obj
```

i Overview | default_report org.eclipse.mat.api:suspects | OQL

SELECT distinct objects classof(obj) FROM "geym.zbase.*" obj

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
class geym.zbase.ch7.heap.Student @ 0x2470dce0	0	0
class geym.zbase.ch7.heap.WebPage @ 0x24705578	0	0
Σ Total: 2 entries		

图 7.42 classof()函数输出

7.5 更精彩的查找：Visual VM 对 OQL 的支持

在第 6 章中，已经简单地介绍了如何通过 Visual VM 查看堆内存快照中的对象信息。但通常堆内存快照十分庞大，快照中的类数量也很多，很难通过浏览的方式找到所需的内容。为此，Visual VM 也和 MAT 一样，提供了对 OQL（对象查询语言）的支持，以方便开发人员在庞大的堆内存数据中，快速定位所需的资源。但不幸的是，MAT 的 OQL 和 Viusal VM 的 OQL 在语法上很不一样，故需要对两者做独立的介绍。

7.5.1 Visual VM 的 OQL 基本语法

Visual VM 的 OQL 语言是一种类似于 SQL 的查询语言，它的基本语法如下：

```
select <JavaScript expression to select>
[ from [instanceof] <class name> <identifier>
[ where <JavaScript boolean expression to filter> ] ]
```

OQL 由 3 个部分组成：select 子句、from 子句和 where 子句。select 子句指定查询结果要显示的内容。from 子句指定查询范围，可指定类名，如 java.lang.String、char[]、[Ljava.io.File;（File 数组）。where 子句用于指定查询条件。

注意：对于 MAT 来说，OQL 的关键字，如 select、from 等可以使用大写，也可以使用小写，但对于 Visual VM 而言，必须统一使用小写。

select 子句和 where 子句支持使用 JavaScript 语法处理较为复杂的查询逻辑，select 子句可

以使用类似 json 的语法输出多个列。from 子句中可以使用 instanceof 关键字，将给定类的子类也包括到输出列表中。

在 Visual VM 的 OQL 中，可以直接访问对象的属性和部分方法。如下例中，直接使用了 String 对象的 count 属性，筛选出长度大于等于 100 的字符串。

```
select s from java.lang.String s where s.count >= 100  (JDK 1.6)
select s from java.lang.String s where s.value.length >= 100 (JDK 1.7)
```

选取长度大于等于 256 的 int 数组。

```
select a from int[] a where a.length >= 256
```

筛选出以“geym”开头的字符串。

```
select {instance: s, content: s.toString()} from java.lang.String s where /^geym.*$/(s.toString())
```

上例中，select 子句使用了 json 语法，指定输出两列为 String 对象以及 String.toString()的输出。where 子句使用正则表达式，指定了符合/^geym.*$/条件的字符串。本例的部分输出数据如下所示：

```
{
 content = geym.zbase.ch7.heap.TraceStudent,
 instance = java.lang.String#924
}

{
 content = geym.zbase.ch7.heap.TraceStudent,
 instance = java.lang.String#1280
}
```

下例筛选出所有的文件路径及文件对象，其中调用了类的 toString()方法。

```
select {content:file.path.toString(),instance:file} from java.io.File file
```

下例使用 instanceof 关键字选取所有的 ClassLoader，包括子类。

```
select cl from instanceof java.lang.ClassLoader cl
```

7.5.2 内置 heap 对象

heap 对象是 Visual VM OQL 的内置对象。通过 heap 对象可以实现一些强大的 OQL 功能。heap 对象的主要方法如下。

- forEachClass()：对每一个 Class 对象执行一个回调操作。它的使用方法类似于 heap.forEachClass(callback)，其中 callback 为 JavaScript 函数。
- findClass()：查找给定名称的类对象，返回类的方法和属性如表 7.4 所示。它的调用方法类似 heap.findClass(className)。
- classes()：返回堆快照中所有的类集合。使用方法如：heap.classes()。
- objects()：返回堆快照中所有的对象集合。使用方法如 heap.objects(clazz, [includeSubtypes], [filter])，其中 clazz 指定类名称，includeSubtypes 指定是否选出子类，filter 为过滤器，指定筛选规则。includeSubtypes 和 filter 可以省略。
- livepaths()：返回指定对象的存活路径。即，显示哪些对象直接或者间接引用了给定对象。它的使用方法如 heap.livepaths(obj)。
- roots()：返回这个堆的根对象。使用方法如 heap.roots()。

表 7.4　使用findClass()返回的Class对象拥有的属性和方法

属性	方法
name：类名称	isSubclassOf()：是否是指定类的子类
superclass：父类	isSuperclassOf()：是否是指定类的父类
statics：类的静态变量的名称和值	subclasses()：返回所有子类
fields：类的域信息	superclasses()：返回所有父类

下例查找 java.util.Vector 类：

```
select heap.findClass("java.util.Vector")
```

查找 java.util.Vector 的所有父类：

```
select heap.findClass("java.util.Vector").superclasses()
```

输出结果如下：

```
java.util.AbstractList
java.util.AbstractCollection
java.lang.Object
```

查找所有在 java.io 包下的对象：

```
select filter(heap.classes(), "/java.io./(it.name)")
```

查找字符串“56”的引用链：

```
select heap.livepaths(s) from java.lang.String s where s.toString()=='56'
```

如下是一种可能的输出结果，其中 java.lang.String#1600 即字符串“56”。它显示了该字符

串被一个 WebPage 对象持有。

```
java.lang.String#1600->geym.zbase.ch7.heap.WebPage#57->java.lang.Object[
]#341->java.util.Vector#11->geym.zbase.ch7.heap.Student#3
```

查找这个堆的根对象：

```
select heap.roots()
```

下例查找当前堆中所有 java.io.File 对象实例，参数 true 表示 java.io.File 的子类也需要被显示：

```
select heap.objects("java.io.File",true)
```

下例访问了 TraceStudent 类的静态成员 webpages 对象：

```
select heap.findClass("geym.zbase.ch7.heap.TraceStudent").webpages
```

说明：本节中部分查询语句使用了 7.3.4 节中产生的堆文件，读者可以先阅读该章节。

7.5.3 对象函数

在 Visual VM 中，为 OQL 语言还提供了一组以对象为操作目标的内置函数。通过这些函数，可以获取目标对象的更多信息。本节主要介绍一些常用的对象函数。

1. classof()函数

返回给定 Java 对象的类。调用方法形如 classof(objname)。返回的类对象有以下属性。

- name：类名称。
- superclass：父类。
- statics：类的静态变量的名称和值。
- fields：类的域信息。

Class 对象拥有以下方法。

- isSubclassOf()：是否是指定类的子类。
- isSuperclassOf()：是否是指定类的父类。
- subclasses()：返回所有子类。
- superclasses()：返回所有父类。

下例将返回所有 Vector 类以及子类的类型：

```
select classof(v) from instanceof java.util.Vector v
```

一种可能的输出如下：

```
java.util.Vector
java.util.Vector
java.util.Stack
```

2. objectid()函数

objectid()函数返回对象的 ID。使用方法如 objectid(objname)。

返回所有 Vector 对象（不包含子类）的 ID：

```
select objectid(v) from java.util.Vector v
```

3. reachables()函数

reachables()函数返回给定对象的可达对象集合。使用方法如 reachables(obj,[exclude])。obj 为给定对象，exclude 指定忽略给定对象中的某一字段的可达引用。

下例返回 WebPage 的可达对象：

```
select {r:toHtml(reachables(s)),url:s.url.toString()} from geym.zbase.ch7.
heap.WebPage s
```

它的部分输出如下：

```
{
r = [ java.lang.String#1432, java.lang.String#1431, char[]#2026, char[]#2027, ],
url = http://www.0.com
}
{
r = [ java.lang.String#1435, java.lang.String#1434, char[]#2030, char[]#2031, ],
url = http://www.1.com
}
```

这里的返回结果是 WebPage.url 和 WebPage.content 两个字段的引用对象。如果使用过滤，要求输出结果中不包含 WebPage.content 字段的引用对象。代码如下：

```
select {r:toHtml(reachables(s,'geym.zbase.ch7.heap.WebPage.content')),url:
s.url.toString()} from geym.zbase.ch7.heap.WebPage s
```

以上查询输出如下：

```
{
r = [ java.lang.String#1431, char[]#2026, ],
url = http://www.0.com
}
{
```

```
r = [ java.lang.String#1434, char[]#2030, ],
url = http://www.1.com
}
```

可以看到，引用对象减少了一半，目前显示的对象都是通过 WebPage.url 字段引用得到的。

4. referrers()函数

referrers()函数返回引用给定对象的对象集合。使用方法如：referrers(obj)。

下例返回了引用表示"http://www.15.com"域名的 WebPage 对象，并且该对象本身也被其他对象引用。

```
select filter(referrers(s),'count(referrers(it))>0') from geym.zbase.ch7.heap.WebPage s where s.url.toString() == "http://www.15.com"
```

它的输出可能如下：

```
java.lang.Object[]#339
java.lang.Object[]#340
```

可以看到有两个数组保存着这个 WebPage 的引用，根据前文描述的该程序的用意，http://www.15.com 也确实应该被 2 名学生访问。在实例页面找到 java.lang.Object[]#339 对象，如图 7.43 所示。

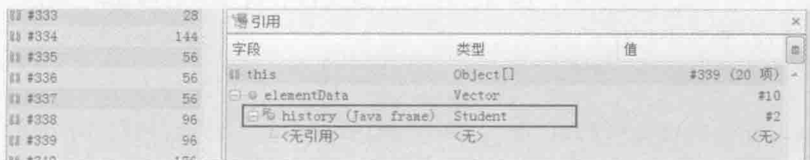


图 7.43 WebPage 最终被 Student 对象引用

下例找出长度为 2，并且至少被两个对象引用的字符串：

```
JDK 1.6 产生的堆
select s.toString() from java.lang.String s where (s.count==2 && count(referrers(s)) >=2)

JDK 1.7 产生的堆
select s.toString() from java.lang.String s where ( s.value != null && s.value.length==2 && count(referrers(s)) >=2)
```

注意： where 子句中使用的逻辑运算符是&&。这是 JavaScript 语法，不能像 SQL 一样使用 AND 操作符。

5. referees()函数

referees()函数返回给定对象的直接引用对象集合，用法形如：referees(obj)。

下例返回了 File 对象的静态成员引用：

```
select referees(heap.findClass("java.io.File"))
```

下例返回 Student 类直接引用的对象：

```
select referees(s) from geym.zbase.ch7.heap.Student s
```

上述查询的返回为：

```
java.util.Vector#9
java.lang.String#1747
java.util.Vector#10
java.lang.String#1748
java.util.Vector#11
java.lang.String#1749
```

可以看到 3 个 Student 对象分别持有一个 Vector 和 String 对象。其中 Vector 对象就是由 hisotry 字段持有，String 对象就是由 name 字段持有。

6. sizeof()函数

sizeof()函数返回指定对象的大小（不包括它的引用对象），即浅堆（Shallow Size）。

注意：sizeof()函数返回对象的大小不包括对象的引用对象。因此，sizeof()的返回值由对象的类型决定，和对象的具体内容无关。

下例返回所有 int 数组的大小以及对象：

```
select {size:sizeof(o),Object:o} from int[] o
```

下例返回所有 Vector 的大小以及对象：

```
select {size:sizeof(o),Object:o} from java.util.Vector o
```

它的输出可能为如下形式：

```
{
Object = java.util.Vector#1,
size = 24.0
}
{
Object = java.util.Vector#2,
```

```
size = 24.0
}
```

可以看到，不论 Vector 集合包含多少对象。Vector 对象所占用的内存大小始终为 24 字节。这是由 Vector 本身的结构决定的，与其内容无关。sizeof()函数就是返回对象的固有大小。

7. rsizeof()函数

rsizeof()函数返回对象以及其引用对象的大小总和，即深堆（Retained Size）。这个数值不仅与类本身的结构有关，还与对象的当前数据内容有关。

下例显示了所有 Vector 对象的 Shallow Size 以及 Retained Size：

```
select {size:sizeof(o),rsize:rsizeof(o)} from java.util.Vector o
```

部分输出可能如下所示：

```
{
rsize = 80.0,
size = 24.0
}
{
rsize = 80.0,
size = 24.0
}
```

注意：resizeof()取得对象以及其引用对象的大小总和。因此，它的返回值与对象的当前数据内容有关。

8. toHtml()函数

toHtml()函数将对象转为 HTML 显示。

下例将 Vector 对象的输出使用 HTML 进行加粗和斜体显示：

```
select "<b><em>"+toHtml(o)+"</em></b>" from java.util.Vector o
```

输出部分结果如图 7.44 所示。直接点击输出对象，可以展示实例页面中的对应对象。

java.util.Vector#1

java.util.Vector#2

java.util.Vector#3

java.util.Vector#4

图 7.44　toHtml()函数输出

7.5.4　集合/统计函数

Visual VM 中还有一组用于集合操作和统计的函数。可以方便地对结果集进行后处理或者统计操作。集合/统计函数主要有 contains()、count()、filter()、length()、map()、max()、min()、sort()、top()等。

1. contains()函数

contains()函数判断给定集合是否包含满足给定表达式的对象。它的使用方法形如 contains(set,boolexpression)。其中 set 为给定集合，boolexpression 为表达式。在 boolexpression 中，可以使用如下 contains()函数的内置对象。

- it：当前访问对象。
- index：当前对象索引。
- array：当前迭代的数组/集合。

下例返回被 File 对象引用的 String 对象集合。首先通过 referrers(s)得到所有引用 String 对象的对象集合。使用 contains()函数及其参数布尔等式表达式 classof(it).name == 'java.io.File')，将 contains()的筛选条件设置为类名是 java.io.File 的对象。

```
select s.toString() from java.lang.String s where contains(referrers(s),
"classof(it).name == 'java.io.File'")
```

以上查询的部分输出结果如下：

```
D:\tools\jdk1.7_40\jre\bin\zip.dll
D:\tools\jdk1.7_40\jre\bin\zip.dll
D:\tools\jdk1.7_40\jre\lib\ext
C:\Windows\Sun\Java\lib\ext
D:\tools\jdk1.7_40\jre\lib\ext\meta-index
```

通过该 OQL，得到了当前堆中所有的 File 对象的文件名称。可以理解为当前 Java 程序通过 java.io.File 获得已打开或持有的所有文件。

2. count()函数

count()函数返回指定集合内满足给定布尔表达式的对象数量。它的基本使用方法如：count(set, [boolexpression])。参数 set 指定要统计总数的集合，boolexpression 为布尔条件表达式，可以省略，但如果指定，count()函数只计算满足表达式的对象个数。在 boolexpression 表达式中，可以使用以下内置对象。

- it：当前访问对象。

- index：当前对象索引。
- array：当前迭代的数组/集合。

下例返回堆中所有 java.io 包中的类的数量，布尔表达式使用正则表达式表示。

```
select count(heap.classes(), "/java.io./(it.name)")
```

下列返回堆中所有类的数量。

```
select count(heap.classes())
```

3. filter()函数

filter()函数返回给定集合中，满足某一个布尔表达式的对象子集合。使用方法形如 filter(set, boolexpression)。在 boolexpression 中，可以使用以下内置对象。

- it：当前访问对象。
- index：当前对象索引。
- array：当前迭代的数组/集合。

下例返回所有 java.io 包中的类。

```
select filter(heap.classes(), "/java.io./(it.name)")
```

下例返回了当前堆中，引用了 java.io.File 对象并且不在 java.io 包中的所有对象实例。首先使用 referrers()函数得到所有引用 java.io.File 对象的实例，接着使用 filter()函数进行过滤，只选取不在 java.io 包中的对象。

```
select filter(referrers(f), "! /java.io./(classof(it).name)") from java.io.File f
```

4. length()函数

length()函数返回给定集合的数量，使用方法形如 length(set)。

下例返回当前堆中所有类的数量。

```
select length(heap.classes())
```

5. map()函数

map()函数将结果集中的每一个元素按照特定的规则进行转换，以方便输出显示。使用方法形如：map(set, transferCode)。set 为目标集合，transferCode 为转换代码。在 transferCode 中可以使用以下内置对象。

- it：当前访问对象。
- index：当前对象索引。

- array：当前迭代的数组/集合。

下例将当前堆中的所有 File 对象进行格式化输出：

```
select map(heap.objects("java.io.File"), "index + '=' + it.path.toString()")
```

输出结果为：

```
0=D:\tools\jdk1.7_40\jre\bin\zip.dll
1=D:\tools\jdk1.7_40\jre\bin\zip.dll
2=D:\tools\jdk1.7_40\jre\lib\ext
3=C:\Windows\Sun\Java\lib\ext
4=D:\tools\jdk1.7_40\jre\lib\ext\meta-index
5=D:\tools\jdk1.7_40\jre\lib\ext
```

注意：map()函数可以用于输出结果的数据格式化。它可以将集合中每一个对象转成特定的输出格式。

6. max()函数

max()函数计算并得到给定集合的最大元素。使用方法为：max(set, [express])。其中 set 为给定集合，express 为比较表达式，指定元素间的比较逻辑。参数 express 可以省略，若省略，则执行数值比较。参数 express 可以使用以下内置对象。

- lhs：用于比较的左侧元素。
- rhs：用于比较的右侧元素。

下例显示了当前堆中最长的 String 长度。对于 JDK 1.6 得到的堆，首先使用 heap.objects() 函数得到所有 String 对象，接着，使用 map()函数将 String 对象集合转为 String 对象的长度集合，最后，使用 max()函数得到集合中的最大元素。对于 JDK 1.7 得到的堆，由于 String 结构发生变化，故通过 String.value 得到字符串长度。

```
JDK 1.6 导出的堆
select max(map(heap.objects('java.lang.String', false), 'it.count'))

JDK 1.7 导出的堆
select max(map(filter(heap.objects('java.lang.String', false),'it.value!=null'),
'it.value.length'))
```

以上 OQL 的输出为最大字符串长度，输出如下：

```
734.0
```

下例取得当前堆的最长字符串。它在 max()函数中设置了比较表达式，指定了集合中对象

的比较逻辑。

```
JDK 1.6 导出的堆
select max(heap.objects('java.lang.String'), 'lhs.count > rhs.count')

JDK 1.7 导出的堆
select max(filter(heap.objects('java.lang.String'),'it.value!=null'), 'lhs.
value.length > rhs.value.length')
```

与上例相比，它得到的是最大字符串对象，而非对象的长度：

```
java.lang.String#908
```

7. min()函数

min()函数计算并得到给定集合的最小元素。使用方法为：min(set, [expression])。其中 set 为给定集合，expression 为比较表达式，指定元素间的比较逻辑。参数 expression 可以省略，若省略，则执行数值比较。参数 expression 可以使用以下内置对象：

- lhs：用于比较的左侧元素
- rhs：用于比较的右侧元素

下例返回当前堆中数组长度最小的 Vector 对象的长度：

```
select min(map(heap.objects('java.util.Vector', false), 'it.elementData.
length'))
```

下例得到数组元素长度最长的一个 Vector 对象：

```
select min(heap.objects('java.util.Vector'), 'lhs.elementData.length >
rhs.elementData.length')
```

8. sort()函数

sort()函数对指定的集合进行排序。它的一般使用方法为：sort(set, expression)。其中，set 为给定集合，expression 为集合中对象的排序逻辑。在 expression 中可以使用以下内置对象：

- lhs：用于比较的左侧元素
- rhs：用于比较的右侧元素

下例将当前堆中的所有 Vector 按照内部数组的大小进行排序：

```
select sort(heap.objects('java.util.Vector'), 'lhs.elementData.length -
rhs.elementData.length')
```

下例将当前堆中的所有 Vector 类（包括子类），按照内部数据长度大小，从小到大排序，

并输出 Vector 对象的实际大小以及对象本身。

```
select map(
    sort(
        heap.objects('java.util.Vector'),
        'lhs.elementData.length - rhs.elementData.length'
    ),
    '{ size: rsizeof(it), obj: it }'
    )
```

上述查询中，首先通过 heap.objects()方法得到所有 Vector 及其子类的实例，接着，使用 sort()函数，通过 Vector 内部数组长度进行排序，最后使用 map()函数对排序后的集合进行格式化输出。

9. top()函数

top()函数返回在给定集合中，按照特定顺序排序的前几个对象。一般使用方法为：top(set, expression,num)。其中 set 为给定集合，expression 为排序逻辑，num 指定输出前几个对象。在 expression 中，可以使用以下内置对象。

- lhs：用于比较的左侧元素。
- rhs：用于比较的右侧元素。

下例显示了长度最长的前 5 个字符串：

```
JDK 1.6 的堆
select top(heap.objects('java.lang.String'), 'rhs.count - lhs.count', 5)

JDK 1.7 的堆
select top(filter(heap.objects('java.lang.String'),'it.value!=null'), 'rhs.
value.length - lhs.value.length', 5)
```

下例显示长度最长的 5 个字符串，输出它们的长度与对象：

```
JDK 1.6 的堆
select map(top(heap.objects('java.lang.String'), 'rhs.count - lhs.count',
5), '{ length: it.count, obj: it }')

JDK 1.7 的堆
select map(top(filter(heap.objects('java.lang.String'),'it.value!=null'),
'rhs.value.length - lhs.value.length', 5), '{ length: it.value.length, obj:
it }')
```

上述查询的部分输出可能如下所示：

```
{
length = 734.0,
obj = java.lang.String#908
}
{
length = 293.0,
obj = java.lang.String#914
}
```

10. sum()函数

sum()函数用于计算集合的累计值。它的一般使用方法为：sum(set,[expression])。其中第一个参数 set 为给定集合，参数 expression 用于将当前对象映射到一个整数，以便用于求和。参数 expression 可以省略，如果省略，则可以使用 map()函数作为替代。

下例计算所有 Student 对象的可达对象的总大小：

```
select sum(map(reachables(p), 'sizeof(it)')) from geym.zbase.ch7.heap.Student p
```

将使用 sum()函数的第 2 个参数 expression 代替 map()函数，实现相同的功能：

```
select sum(reachables(p), 'sizeof(it)') from geym.zbase.ch7.heap.Student p
```

11. unique()函数

unique()函数将除去指定集合中的重复元素，返回不包含重复元素的集合。它的一般使用方法形如 unique(set)。

下例返回当前堆中，有多个不同的字符串：

```
select count(unique(map(heap.objects('java.lang.String'), 'it.value')))
```

7.5.5 程序化 OQL 分析 Tomcat 堆

Visual VM 不仅支持在 OQL 控制台上执行 OQL 查询语言，也可以通过其 OQL 相关的 JAR 包，将 OQL 查询程序化，从而获得更加灵活的对象查询功能，实现堆快照分析的自动化。

【示例 7-7】这里以分析 Tomcat 堆溢出文件为例，展示程序化 OQL 带来的便利。

在进行 OQL 开发前，工程需要引用 Visual VM 安装目录下 JAR 包，如图 7.45 所示。

在本示例中，加入如图 7.46 所示的 JAR 包。



图 7.45　Viusal VM 中 OQL 相关 JAR 包



图 7.46　加入工程的 Viusal VM 的 JAR 包

对于给定的 Tomcat 堆溢出 Dump 文件，这里将展示如何通过程序，计算 Tomcat 平均每秒产生的 session 数量，代码如下：

```
public class AveLoadTomcatOOM {
    public static final String dumpFilePath="d:/tmp/tomcat_oom/tomcat.hprof";

    public static void main(String args[]) throws Exception{
        OQLEngine engine;
        final List<Long> creationTimes=new ArrayList<Long>(10000);
        engine=new OQLEngine(HeapFactory.createHeap(new File(dumpFilePath)));
        String query="select s.creationTime from org.apache.catalina.session.StandardSession s";
        engine.executeQuery(query, new OQLEngine.ObjectVisitor(){
            public boolean visit(Object obj){
                creationTimes.add((Long)obj);
                return false;
            }
        });

        Collections.sort(creationTimes);

        long min=creationTimes.get(0)/1000;
        long max=creationTimes.get(creationTimes.size()-1)/1000;

        System.out.println("平均压力："+creationTimes.size()*1.0/(max-min)+"次/秒");
```

```
22     }
23 }
```

上述代码第 8 行，通过 OQL 语句得到所有 session 的创建时间，在第 18、19 行获得所有 session 中最早创建和最晚创建的 session 时间，在第 21 行计算整个时间段内的平均 session 创建速度。

运行上述代码，得到输出如下：

```
平均压力：311.34375 次/秒
```

使用这种方式可以做到堆转存文件的全自动化分析，并将结果导出到给定文件，当有多个堆转存文件需要分析时，有着重要的作用。

除了使用以上方式外，Visual VM 的 OQL 控制台也支持直接使用 JavaScript 代码进行编程，如下代码实现了相同功能：

```
var sessions=toArray(heap.objects("org.apache.catalina.session.StandardSession"));
var count=sessions.length;
var createtimes=new Array();
for(var i=0;i<count;i++){
    createtimes[i]=sessions[i].creationTime;
}
createtimes.sort();
var min=createtimes[0]/1000;
var max=createtimes[count-1]/1000;
count/(max-min)+"次/秒"
```

图 7.47 显示了在 OQL 控制台中，执行上述脚本以及输出结果。

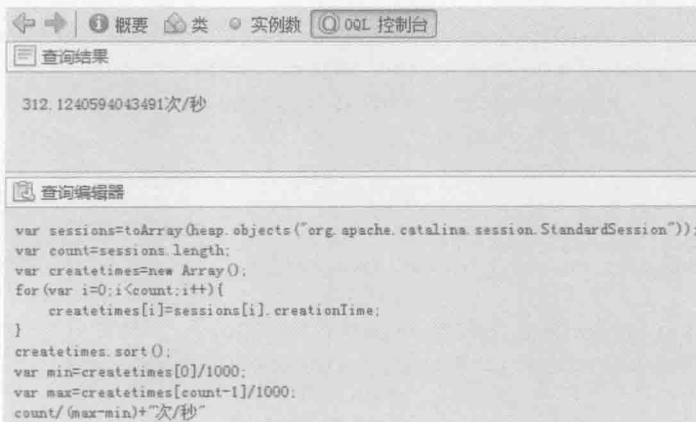


图 7.47　通过 Javascript 计算 Tomcat 中 session 的平均创建速度

细心的读者可能会发现，这个结果和使用 Java 访问 Dump 文件时的结果有所差异，这是因为 JavaScript 是弱类型语言，在处理整数除法时和 Java 有所不同，读者可以自行研究，在此不予展开讨论。

Visual VM 的 OQL 是非常灵活的，除了上述使用 JavaScript 风格外，也可以使用如下函数式编程风格计算：

```
count(heap.objects('org.apache.catalina.session.StandardSession'))/
(
max(map(heap.objects('org.apache.catalina.session.StandardSession'),'it.
creationTime'))/1000-
min(map(heap.objects('org.apache.catalina.session.StandardSession'),'it.
creationTime'))/1000
)
```

上述代码使用了 count()、min()、max()、map()等函数，共同完成了平均值的计算。执行上述代码，输出如下：

```
312.1240594043491
```

7.6　小结

本章主要介绍了 Java 堆的分析方法。首先，介绍了几种常见的 Java 内存溢出现象及解决思路。其次，探讨了 java.lang.String 类的特点，以及在 JDK 1.6 和 JDK 1.7 中的改进与区别。本章最重要的部分是使用 MAT 和 Visual VM 对 Java 堆进行解读和分析，对于使用 Visual VM 分析 Java 堆，本章不仅给出了基于 OQL 查询的分析策略，还提出了一种程序化的分析方法。

8

第 8 章 锁与并发

随着多核计算的兴起，适应于多核计算的多线程开发模式得到了越来越普遍的应用。Java 虚拟机对多线程开发有着很好的支持，其中，一个重要的元素就是对“锁”的实现和优化。本章将重点介绍 Java 虚拟机内部对“锁”的实现、优化以及应用软件对“锁”的优化方法。但本章只对多线程基础做简单介绍，如果读者对多线程开发的基础感兴趣，建议参考其他相关书籍。

本章涉及的主要知识点有：

- 理解线程安全的重要性。
- “锁”在虚拟机内的基本实现方式。
- 应用层对“锁”进行优化的一般方法和思路。
- 无锁计算的方法和原理。
- 理解 Java 虚拟机内存模型。

8.1 安全就是锁存在的理由：锁的基本概念和实现

锁是多线程软件开发的必要工具之一，它的基本作用是保护临界区资源不会被多个线程同时访问而受到破坏。如果由于多线程访问造成对象数据的不一致，那么系统运行将会得到错误的结果。通过锁，可以让多个线程排队，一个一个地进入临界区访问目标对象，使目标对象的状态总是保持一致，这也就是锁存在的价值。

8.1.1 理解线程安全

通过锁，可以实现线程安全，对于线程安全简单的理解，就是在多线程环境下，无论多个线程如何访问目标对象，目标对象的状态应该始终是保持一致的，线程的行为也总是正确的。

【示例 8-1】下面以一个简单的示例，说明线程安全的概念。

如图 8.1 所示，线程 A 和线程 B 在数据库中分别读入两条学生成绩记录，线程 A 读入小明考了 98 分，线程 B 读入小王考了 77 分。现在需要将从数据库里得到的数据保存到对象实例 S 上，再进行其他相应的业务逻辑处理。此时，对象实例 S 就是临界区资源。如果没有锁对它进行保护，任由线程 A 和 B 随意处理，由于线程间的无序性访问，一种可能的访问结果是，线程 A 将学生名“小明” 赋予对象 S，接着线程 B 将学生名“小王”赋予对象 S，覆盖了线程 A 的操作。然后，线程 B 将成绩 77 赋予对象 S，最后线程 A 将成绩 98 赋予对象 S，覆盖了线程 B 的操作。这一组操作得到的结果是对象 S 中保存了部分小明的数据（成绩），部分小王的数据（学生名），显然这样一个对象是没有任何意义的，也就是对象处于一种不一致的状态，这也正是线程不安全导致的恶果。

要处理这个问题就可以使用锁来解决。对于对象 S 的所有操作使用锁进行控制，每一次只允许一个线程对其操作，如果线程 A 先获得锁，那么线程 A 将完成它对对象 S 的所有处理，最后释放锁。而线程 B 由于没能请求到锁，就会进行等待，直到线程 A 释放了锁，线程 B 才得以进入。在这种情况下，只有在被锁保护的代码段内，对象的状态会出现短暂的不一致（幸运的是，这种状态被锁保护，因此其他线程也无法观察到这种状态），但只要线程 A 或者线程 B 完成了它的工作，对象 S 的状态就是一致的，即对象 S 保存的数据不是小明的就是小王的，而不是两者的混合体。

数据的不一致不仅仅会使得程序给出错误的结果，也可能导致程序异常崩溃。

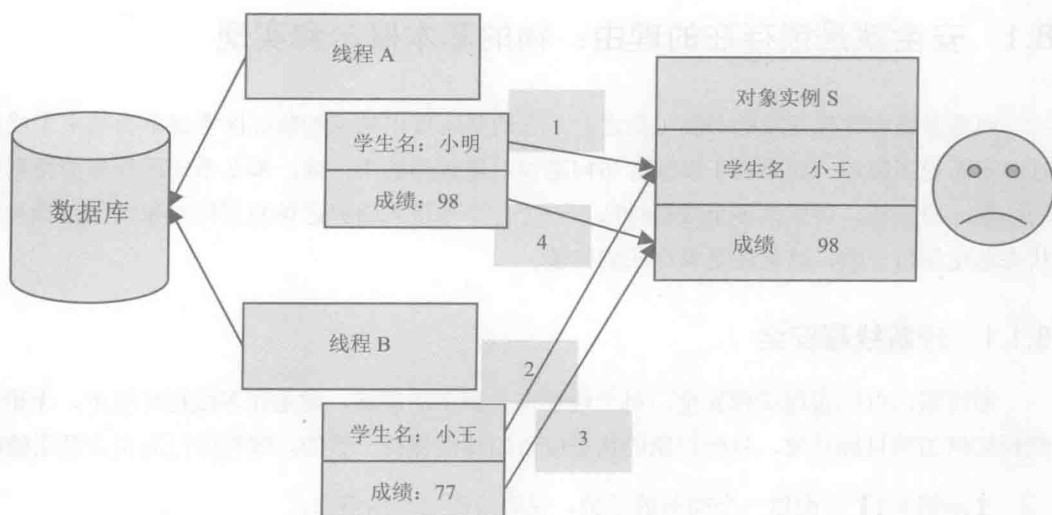


图 8.1　线程不安全导致的数据不一致

【示例 8-2】一个典型的例子就是将 ArrayList 在多线程下使用，以下代码演示了这个错误。

```
public class ThreadUnSafe {
    public static List<Integer> numberList =new ArrayList<Integer>();
    public static class AddToList implements Runnable{
        int startnum=0;
        public AddToList(int startnumber){
            startnum=startnumber;
        }
        @Override
        public void run() {
            int count=0;
            while(count<1000000){
                numberList.add(startnum);
                startnum+=2;
                count++;
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        Thread t1=new Thread(new AddToList(0));
        Thread t2=new Thread(new AddToList(1));
        t1.start();
```

```
        t2.start();
    }
}
```

上述代码中，两个线程 t1 和 t2 同时向 numberList 增加数据。由于 ArrayList 不是线程安全的，因此程序运行后，很有可能抛出以下错误（也有可能不出错）：

```
Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException:
1215487
    at java.util.ArrayList.add(ArrayList.java:441)
    at
geym.zbase.ch8.threadsafe.ThreadUnSafe$AddToList.run(ThreadUnSafe.java:17)
    at java.lang.Thread.run(Thread.java:724)
```

出现这个问题，是因为两个线程同时对 ArrayList 进行写操作，破坏了 ArrayList 内部数据的一致性，导致其中一个线程访问了错误的数组索引。一个简单的修正方法就是使用 Vector 代替 ArrayList。在 Vector 的实现中，使用了内部锁对 List 对象进行控制，如下所示：

```
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}
```

关键字 synchronized 保证了每次只有一个线程可以访问对象实例，确保了多线程环境中对象内部数据的一致性。

8.1.2 对象头和锁

在 Java 虚拟机的实现中每个对象都有一个对象头，用于保存对象的系统信息。对象头中有一个称为 Mark Word 的部分，它是实现锁的关键。在 32 位系统中，Mark Word 为一个 32 位的数据，在 64 位系统中，它占 64 位。它是一个多功能的数据区，可以存放对象的哈希值、对象年龄、锁的指针等信息。一个对象是否占用锁，占有哪个锁，就记录在这个 Mark Word 中。

以 32 位系统为例，普通对象的对象头如下所示：

```
hash:25 ------------>| age:4    biased_lock:1 lock:2
```

它表示 Mark Word 中有 25 位比特表示对象的哈希值，4 位比特表示对象的年龄，1 位比特表示是否为偏向锁，2 位比特表示锁的信息。

对于偏向锁的对象，它的格式如下：

```
[JavaThread* | epoch | age | 1 | 01]
```

前 23 位表示持有偏向锁的线程，后续 2 位比特表示偏向锁的时间戳（epoch），4 位比特表示对象年龄，年龄后 1 位比特固定为 1，表示偏向锁，最后 2 位为 01 表示可偏向/未锁定。

当对象处于轻量级锁定时，其 Mark Word 如下（00 表示最后 2 位的值）：

```
[ptr             | 00]  locked
```

此时，它指向存放在获得锁的线程栈中的该对象真实对象头。

当对象处于重量级锁定时，其 Mark Word 如下：

```
[ptr             | 10]  monitor
```

此时，最后 2 位为 10，整个 Mark Word 表示指向 Monitor 的指针。

当对象处于普通的未锁定状态时，其格式如下：

```
[header      | 0 | 01]  unlocked
```

前 29 位表示对象的哈希值、年龄等信息。倒数第 3 位为 0，最后两位为 01，表示未锁定。可以发现，最后两位的值和偏向状态时是一样的，此时，虚拟机正是通过倒数第 3 位比特来区分是否为偏向锁。

8.2 避免残酷的竞争：锁在 Java 虚拟机中的实现和优化

在了解了对象头 Mark Word 的基本概念后，就可以深入虚拟机内部，一探虚拟机对锁的实现方式。在多线程程序中，线程之间的竞争是不可避免的，而且是一种常态，如何使用更高的效率处理多线程的竞争，是 Java 虚拟机一项重要的使命。如果将所有的线程竞争都交由操作系统处理，那么并发性能将是非常低下的，为此，虚拟机在操作系统层面挂起线程之前，会先尽一切可能在虚拟机层面上解决竞争关系，尽可能避免真实的竞争发生。同时，在竞争不激烈的场合，也会试图消除不必要的竞争。本节将介绍实现这些手段的方法，包括偏向锁、轻量级锁、自旋锁、锁消除、锁膨胀等。

8.2.1 偏向锁

偏向锁是 JDK 1.6 提出的一种锁优化方式。其核心思想是，如果程序没有竞争，则取消之

前已经取得锁的线程同步操作。也就说，若某一锁被线程获取后，便进入偏向模式，当线程再次请求这个锁时，无需再进行相关的同步操作，从而节省了操作时间。如果在此之间有其他线程进行了锁请求，则锁退出偏向模式。在 JVM 中使用-XX:+UseBiasedLocking 可以设置启用偏向锁。

当锁对象处于偏向模式时，对象头会记录获得锁的线程：

```
[JavaThread* | epoch | age | 1 | 01]
```

这样，当该线程再次尝试获得锁时，通过 Mark Word 的线程信息就可以判断当前线程是否持有偏向锁。

【示例 8-3】下面这段代码可以展示使用偏向锁之后的性能提升，在笔者的测试中，使用偏向锁简化锁的处理流程，可以获得大约 20%左右的性能提升。

```
public class Biased {
    public static List<Integer> numberList =new Vector<Integer>();
    public static void main(String[] args) throws InterruptedException {
        long begin=System.currentTimeMillis();
        int count=0;
        int startnum=0;
        while(count<10000000){
            numberList.add(startnum);
            startnum+=2;
            count++;
        }
        long end=System.currentTimeMillis();
        System.out.println(end-begin);
    }
}
```

上述代码使用一个线程对 Vector 进行写入操作，由于对 Vector 的访问，其内部都是用同步锁控制，故每次 add()操作都会请求 numberList 对象的锁。使用以下参数执行这段程序：

```
-XX:+UseBiasedLocking  -XX:BiasedLockingStartupDelay=0  -client  -Xmx512m
-Xms512m
```

程序输出结果如下：

```
394
```

这说明程序在 394 毫秒内完成所有工作。参数中的-XX:BiasedLockingStartupDelay 表示虚拟机在启动后，立即启用偏向锁。如不设置该参数，虚拟机默认会在启动后 4 秒后，才启用偏

向锁，考虑到程序运行时间较短，故做此设置，尽早启用偏向锁。

若禁用偏向锁，则只需使用如下参数启动程序：

```
-XX:-UseBiasedLocking -client -Xmx512m -Xms512m
```

程序输出结果如下：

```
539
```

可以看到，偏向锁在少竞争的情况下，对系统性能有一定帮助。

偏向锁在锁竞争激烈的场合没有太强的优化效果，因为大量的竞争会导致持有锁的线程不停地切换，锁也很难一直保持在偏向模式，此时，使用锁偏向不仅得不到性能的优化，反而有可能降低系统性能。因此，在激烈竞争的场合，可以尝试使用-XX:-UseBiasedLocking 参数禁用偏向锁。

8.2.2 轻量级锁

如果偏向锁失败，Java 虚拟机会让线程申请轻量级锁。轻量级锁在虚拟机内部，使用一个称为 BasicObjectLock 的对象实现，这个对象内部由一个 BasicLock 对象和一个持有该锁的 Java 对象指针组成。BasicObjectLock 对象放置在 Java 栈的栈帧中。在 BasicLock 对象内部还维护着 displaced_header 字段，它用于备份对象头部的 Mark Word。

当一个线程持有一个对象的锁时，对象头部 Mark Word 如下所示：

```
[ptr             | 00]  locked
```

末尾两位比特为 00，整个 Mark Word 为指向 BasicLock 对象的指针。由于 BasicObjectLock 对象在线程栈中，因此该指针必然指向持有该锁的线程栈空间。当需要判断某一线程是否持有该对象锁时，也只需简单地判断对象头的指针是否在当前线程的栈地址范围内即可。同时，BasicLock 对象的 displaced_header 字段，备份了原对象的 Mark Word 内容。BasicObjectLock 对象的 obj 字段则指向该对象。

在虚拟机的实现中，有关轻量级加锁的代码实现可读性较好，这里给出其实现核心代码，帮助读者理解。

```
markOop mark = obj->mark();
lock->set_displaced_header(mark);
if (mark == (markOop) Atomic::cmpxchg_ptr(lock, obj()->mark_addr(), mark)) {
  TEVENT (slow_enter: release stacklock) ;
```

```
    return ;
  }
```

首先，BasicLock 通过 set_displaced_header()方法备份了原对象的 Mark Word。接着，使用 CAS 操作，尝试将 BasicLock 的地址复制到对象头的 Mark Word。如果复制成功，那么加锁成功，否则认为加锁失败。如果加锁失败，那么轻量级锁就有可能被膨胀为重量级锁。

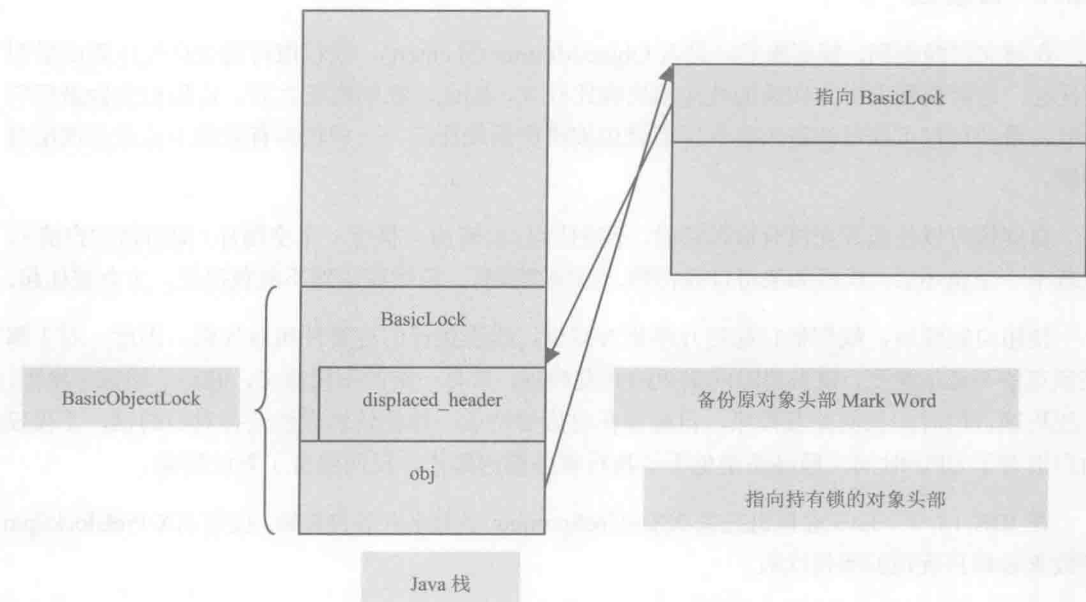


图 8.2　轻量级锁示意图

8.2.3　锁膨胀

当轻量级锁失败，虚拟机就会使用重量级锁。在使用重量级锁时，对象的 Mark Word 如下：

```
[ptr             | 10]  monitor
```

末尾的 2 比特标记位被置为 10。整个 Mark Word 表示指向 monitor 对象的指针。在轻量级锁处理失败后，虚拟会执行以下操作：

```
lock->set_displaced_header(markOopDesc::unused_mark());
ObjectSynchronizer::inflate(THREAD, obj())->enter(THREAD);
```

第 1 步是废弃前面 BasicLock 备份的对象头信息。第 2 步则正式启用重量级锁。启用过程分为两步：首先通过 inflate()方法进行锁膨胀，其目的是获得对象的 ObjectMonitor；然后使用

enter()方法尝试进入该锁。

在 enter()方法调用中，线程很可能会在操作系统层面被挂起。如果这样，线程间切换和调度的成本就会比较高。

8.2.4 自旋锁

在前文已经提到，锁膨胀后，进入 ObjectMonitor 的 enter()，线程很可能会在操作系统层面被挂起，这样线程上下文切换的性能损失就比较大。因此，在锁膨胀之后，虚拟机会做最后的争取，希望线程可以尽快进入临界区而避免被操作系统挂起。一种较为有效的手段就是使用自旋锁。

自旋锁可以使线程在没有取得锁时，不被挂起，而转而去执行一个空循环（即所谓的自旋），在若干个空循环后，线程如果可以获得锁，则继续执行。若线程依然不能获得锁，才会被挂起。

使用自旋锁后，线程被挂起的几率相对减少，线程执行的连贯性相对加强。因此，对于那些锁竞争不是很激烈，锁占用时间很短的并发线程，具有一定的积极意义，但对于锁竞争激烈，单线程锁占用时间长的并发程序，自旋锁在自旋等待后，往往依然无法获得对应的锁，不仅仅白白浪费了 CPU 时间，最终还是免不了执行被挂起的操作，反而浪费了系统资源。

在 JDK 1.6 中，Java 虚拟机提供-XX:+UseSpinning 参数来开启自旋锁，使用-XX:PreBlockSpin 参数来设置自旋锁的等待次数。

在 JDK 1.7 中，自旋锁的参数被取消，虚拟机不再支持由用户配置自旋锁。自旋锁总是会执行，自旋次数也由虚拟机自行调整。

8.2.5 锁消除

锁消除是 Java 虚拟机在 JIT 编译时，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁。通过锁消除，可以节省毫无意义的请求锁时间。

说到这里，细心的读者可能会产生疑问，如果不可能存在竞争，为什么程序员还要加上锁呢？这是因为在 Java 软件开发过程中，开发人员必然会使用一些 JDK 的内置 API，比如 StringBuffer、Vector 等。这些常用的工具类可能会被大面积地使用。虽然这些工具类本身可能有对应的非线程安全版本，但是开发人员也很有可能在完全没有多线程竞争的场合使用它们。

在这种情况下，这些工具类内部的同步方法就是不必要的。虚拟机可以在运行时，基于逃逸分析技术，捕获到这些不可能存在竞争却有申请锁的代码段，并消除这些不必要的锁，从而

提高系统性能。

【示例 8-4】比如，下面代码中 sb 变量的作用域仅限于方法体内部，不可能逃逸出该方法，因此它就不可能被多个线程同时访问。

```
public class LockEliminate {
    private static final int CIRCLE = 2000000;
    public static void main(String args[]) throws InterruptedException {
        long start = System.currentTimeMillis();
        for (int i = 0; i < CIRCLE; i++) {
            craeteStringBuffer("JVM", "Diagnosis");
        }
        long bufferCost = System.currentTimeMillis() - start;
        System.out.println("craeteStringBuffer: " + bufferCost + " ms");
    }

    public static String craeteStringBuffer(String s1, String s2) {
        StringBuffer sb = new StringBuffer();
        sb.append(s1);
        sb.append(s2);
        return sb.toString();
    }
}
```

逃逸分析和锁消除分别可以使用参数-XX:+DoEscapeAnalysis 和-XX:+EliminateLocks 开启（锁消除必须工作在-server 模式下）。

使用以下参数执行这段代码：

```
-server -XX:+DoEscapeAnalysis -XX:-EliminateLocks -Xcomp -XX:-BackgroundCompilation
-XX:BiasedLockingStartupDelay=0
```

上述参数关闭了锁消除，因此，每次 append()操作都会进行锁的申请。执行后，程序输出如下：

```
craeteStringBuffer: 189 ms
```

如果开启锁消除，即使用以下参数执行代码：

```
-server -XX:+DoEscapeAnalysis -XX:+EliminateLocks -Xcomp -XX:-BackgroundCompilation
-XX:BiasedLockingStartupDelay=0
```

则程序的输出如下：

```
craeteStringBuffer: 158 ms
```

可以看到，使用锁消除后，性能有了较为明显的改善。根据前文对偏向锁的介绍，可以知道，偏向锁本身简化了锁的获取，其性能较好。本例中，使用-XX:BiasedLockingStartupDelay参数迫使偏向锁在启动时就生效，即便如此，性能也不如锁消除后的代码。如果在本次实验中不启用偏向锁，那么性能差距会更大，有兴趣的读者可以自行尝试。

8.3 应对残酷的竞争：锁在应用层的优化思路

上一节中主要介绍了虚拟机内部对锁的优化与实现。在实际软件开发过程中，如果在应用层能合理地进行锁的优化，也对系统性能有积极作用。本节将主要介绍一些从应用角度对锁进行优化的方法和思路。

8.3.1 减少锁持有时间

对于使用锁进行并发控制的应用程序而言，在锁竞争过程中，单个线程对锁的持有时间与系统性能有着直接的关系。如果线程持有锁的时间很长，那么相对的，锁的竞争程度也就越激烈。因此，在程序开发过程中，应该尽可能地减少对某个锁的占有时间，以减少线程间互斥的可能。以下面的代码段为例：

```
public synchronized void syncMethod(){
    othercode1();
    mutextMethod();
    othercode2();
}
```

syncMethod()方法中，假设只有 mutextMethod()方法是有同步需要的，而 othercode1()和othercode2()并不需要做同步控制。如果 othercode1()和 othercode2()分别是重量级的方法，则会花费较长的 CPU 时间。此时，如果在并发量较大，使用这种对整个方法做同步的方案，会导致等待线程大量增加。因为一个线程，在进入该方法时获得内部锁，只有在所有任务都执行完后，才会释放锁。

一个较为优化的解决方案是，只在必要时进行同步，这样就能明显减少线程持有锁的时间，提高系统的吞吐量。

```
public void syncMethod2(){
    othercode1();
    synchronized(this){
        mutextMethod();
```

```
    }
    othercode2();
}
```

在改进的代码中，只针对 mutextMethod()方法做了同步，锁占用的时间相对较短，因此能有更高的并行度。这种技术手段在 JDK 的源码包中也可以很容易地找到，比如处理正则表达式的 Pattern 类：

```
public Matcher matcher(CharSequence input) {
    if (!compiled) {
        synchronized(this) {
        if (!compiled)
            compile();
        }
    }
    Matcher m = new Matcher(this, input);
    return m;
}
```

matcher()方法有条件地进行锁申请，只有在表达式未编译时，进行局部的加锁，这种处理方式大大提高了 matcher()方法的执行效率和可靠性。

注意：减少锁的持有时间有助于降低锁冲突的可能性，进而提升系统的并发能力。

8.3.2　减小锁粒度

减小锁粒度也是一种削弱多线程锁竞争的有效手段。这种技术典型的使用场景就是 ConcurrentHashMap 类的实现。对一个普通的集合对象的多线程同步来说，最常使用的方式就是对 get()和 add()方法进行同步。每当对集合进行 add()操作或者 get()操作时，总是获得集合对象的锁。因此，事实上没有两个线程可以做到真正的并发，任何线程在执行这些同步方法时，总要等待前一个线程执行完毕。在高并发时，激烈的锁竞争会影响系统的吞吐量。

作为 JDK 并发包中重要的成员 ConcurrentHashMap 类，很好地使用了拆分锁对象的方式提高 ConcurrentHashMap 的吞吐量。ConcurrentHashMap 将整个 HasmMap 分成若干个段（Segment），每个段都是一个子 HashMap。

如果需要在 ConcurrentHashMap 中增加一个新的表项，并不是将整个 HashMap 加锁，而是首先根据 hashcode 得到该表项应该被存放到哪个段中，然后对该段加锁，并完成 put()操作。在多线程环境中，如果多个线程同时进行 put()操作，只要被加入的表项不存放在同一个段中，则

线程间便可以做到真正的并行。

默认情况下，ConcurrentHashMap 拥有 16 个段，因此，如果够幸运的话，ConcurrentHashMap 可以同时接受 16 个线程同时插入（如果都插入不同的段中），从而大大提供其吞吐量。如图 8.3 所示，显示了 6 个线程同时对 ConcurrentHashMap 进行访问，此时，线程 1、2、3 分别访问段 1、2、3，由于段 1、2、3 使用独立的锁保护，因此，3 个线程可以同时访问 ConcurrentHashMap，而线程 4、5、6 也需要访问段 1、2、3，则必须等待前面的线程结束访问才能进入 ConcurrentHashMap。

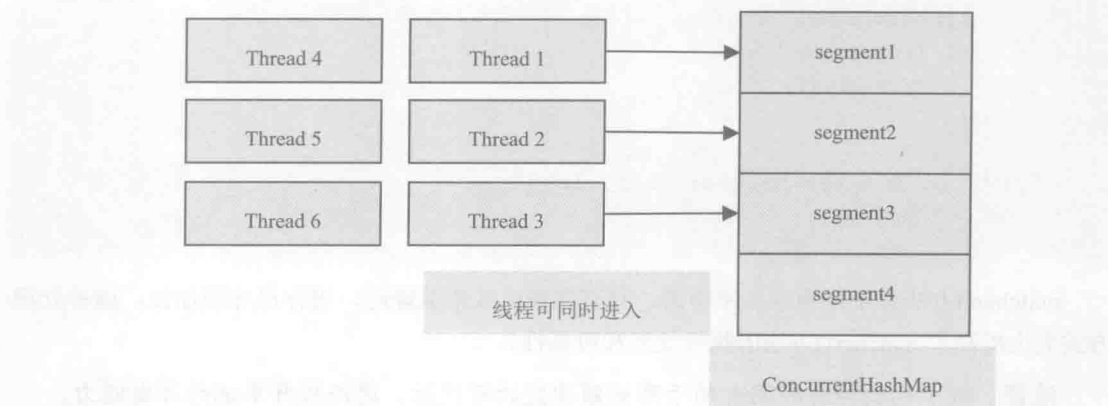


图 8.3　ConcurrentHashMap 实现的示意图

但是，减少锁粒度会引入一个新的问题，即：当系统需要取得全局锁时，其消耗的资源会比较多。仍然以 ConcurrentHashMap 类为例，虽然其 put()方法很好地分离了锁，但是当试图访问 ConcurrentHashMap 全局信息时，就会需要同时取得所有段的锁方能顺利实施。比如 ConcurrentHashMap 的 size()方法，它将返回 ConcurrentHashMap 的有效表项的数量，即 ConcurrentHashMap 的全部有效表项之和。要获取这个信息需要取得所有子段的锁，因此，其 size()方法的部分代码如下：

```
sum = 0;
for (int i = 0; i < segments.length; ++i)                //对所有的段加锁
   segments[i].lock();
for (int i = 0; i < segments.length; ++i)                //统计总数
   sum += segments[i].count;
for (int i = 0; i < segments.length; ++i)                //释放所有的锁
   segments[i].unlock();
```

可以看到在计算总数时，先要获得所有段的锁，然后再求和。但是，ConcurrentHashMap

的 size()方法并不总是这样执行，事实上，size()方法会先使用无锁的方式求和，如果失败才会尝试这种加锁的方法。但不管怎么说，在高并发场合 ConcurrentHashMap 的 size()的性能依然要差于同步的 HashMap。

注意： 所谓减少锁粒度，就是指缩小锁定对象的范围，从而减少锁冲突的可能性，进而提高系统的并发能力。

8.3.3 锁分离

锁分离是减小锁粒度的一个特例，它依据应用程序的功能特点，将一个独占锁分成多个锁。一个典型的案例就是 java.util.concurrent.LinkedBlockingQueue 的实现。

在 LinkedBlockingQueue 的实现中，take()函数和 put()函数分别实现了从队列中取得数据和往队列中增加数据的功能。虽然两个函数都对当前队列进行了修改操作，但由于 LinkedBlockingQueue 是基于链表的，因此，两个操作分别作用于队列的前端和尾端，从理论上说，两者并不冲突。图 8.4 显示了 LinkedBlockingQueue 的 take()和 put()操作，可以看到，在进行 take()和 put()操作时，两者并无冲突。

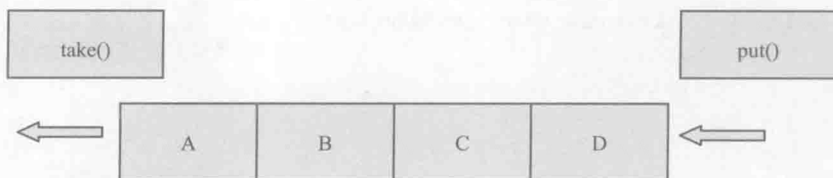


图 8.4　take()和 put()操作并无冲突

如果使用独占锁，则要求在两个操作进行时获取当前队列的独占锁，那么 take()和 put()操作就不可能真正并发，在运行时，它们会彼此等待对方释放锁资源。在这种情况下，锁竞争会相对比较激烈，从而影响程序在高并发时的性能。

因此，在 JDK 的实现中，并没有采用这样的方式，取而代之的是两把不同的锁，分离了 take()和 put()操作。

```
/** Lock held by take, poll, etc */
private final ReentrantLock takeLock = new ReentrantLock();
                                    //take()函数需要持有 takeLock
/** Wait queue for waiting takes */
private final Condition notEmpty = takeLock.newCondition();
/** Lock held by put, offer, etc */
private final ReentrantLock putLock = new ReentrantLock();
```

```
                                                //put()函数需要持有 putLock
    /** Wait queue for waiting puts */
    private final Condition notFull = putLock.newCondition();
```

以上代码片段，定义了 takeLock 和 putLock，它们分别在 take()操作和 put()操作中使用。因此，take()函数和 put()函数就相互独立，它们之间不存在锁竞争关系。只需要在 take()和 take()间、put()和 put()间分别对 takeLock 和 putLock 进行竞争。从而，削弱了锁竞争的可能性。

函数 take()的实现如下，笔者在代码中给出了详细的注释，故不在正文中做进一步说明。

```
public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();                //不能有两个线程同时取数据
    try {
        try {
            while (count.get() == 0)              //如果当前没有可用数据，一直等待
                notEmpty.await();                 //等待，put()操作的通知
        } catch (InterruptedException ie) {
            notEmpty.signal();                    //通知其他未中断的线程
            throw ie;
        }

        x = extract();                            //取得第一个数据
        c = count.getAndDecrement();              //数量减 1，原子操作，因为会和 put()
//函数同时访问 count。注意：变量 c 是
//count 减 1 前的值
        if (c > 1)
            notEmpty.signal();                    //通知其他 take()操作
    } finally {
        takeLock.unlock();                        //释放锁
    }
    if (c == capacity)
        signalNotFull();                          //通知 put()操作，已有空余空间
    return x;
}
```

函数 put()的实现如下：

```
public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
```

```
        int c = -1;
        final ReentrantLock putLock = this.putLock;
        final AtomicInteger count = this.count;
        putLock.lockInterruptibly();                //不能有两个线程同时进行 put()
        try {
            try {
                while (count.get() == capacity)     //如果队列已经满了
                    notFull.await();                //等待
            } catch (InterruptedException ie) {
                notFull.signal();                   //通知未中断的线程
                throw ie;
            }
            insert(e);                              //插入数据
            c = count.getAndIncrement();            //更新总数，变量 c 是 count 加 1 前的值
            if (c + 1 < capacity)
                notFull.signal();                   //有足够的空间，通知其他线程
        } finally {
            putLock.unlock();                       //释放锁
        }
        if (c == 0)
            signalNotEmpty();                       //插入成功后，通知 take() 操作取数据
    }
```

通过 takeLock 和 putLock 两把锁，LinkedBlockingQueue 实现了取数据和写数据的分离，使两者在真正意义上成为可并发的操作。

8.3.4　锁粗化

通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽量短，即在使用完公共资源后，应该立即释放锁。只有这样，等待在这个锁上的其他线程才能尽早地获得资源执行任务。但是，凡事都有一个度，如果对同一个锁不停地进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化。

为此，虚拟机在遇到一连串连续地对同一锁不断进行请求和释放的操作时，便会把所有的锁操作整合成对锁的一次请求，从而减少对锁的请求同步次数。这个操作叫做锁的粗化。比如代码段：

```
public void demoMethod(){
    synchronized(lock){
        //do sth.
```

```
    }
    //做其他不需要同步的工作，但能很快执行完毕
    synchronized(lock){
        //do sth.
    }
}
```

会被整合成如下形式：

```
public void demoMethod(){
        //整合成一次锁请求
    synchronized(lock){
        //do sth.
        //做其他不需要同步的工作，但能很快执行完毕
    }
}
```

在软件开发过程中，开发人员也应该有意识地在合理的场合进行锁的粗化，尤其当在循环内请求锁时。

【示例 8-5】以下是一个循环内请求锁的例子。

```
for(int i=0;i<CIRCLE;i++){
    synchronized(lock){

    }
}
```

以上代码在每一个循环时，都对同一个对象申请锁。此时，应该将锁粗化成：

```
synchronized(lock){
for(int i=0;i<CIRCLE;i++){

    }
}
```

显而易见，第 1 种情况会对锁进行大量的请求，而第 2 种情况只进行一次锁请求，因此，后者的性能会远远高于前者，随着循环次数的增加，性能差距也会越来越明显。

注意： 性能优化就是根据运行时的真实情况对各个资源点进行权衡折中的过程。锁粗化的思想和减少锁持有时间是相反的，但在不同的场合，它们的效果并不相同。开发人员需要根据实际情况进行权衡。此外，前文提到的偏向锁、自旋锁，作

为虚拟机内部的锁优化策略，它们也不是绝对地可以提高系统性能，对锁的优化，还是需要做更多的权衡和思考。

8.4　无招胜有招：无锁

为了确保程序和数据的线程安全，使用“锁”是最直观的一种方式。但是，在高并发时，对“锁”的激烈竞争可能会成为系统瓶颈。为此，开发人员可以使用一种称为非阻塞同步的方法。这种方法不需要使用“锁”（因此称之为“无锁”），但是依然能确保数据和程序在高并发环境下保持多线程间的一致性。本节主要介绍这种无锁的同步方法的实现方式及其使用方法。

8.4.1　理解 CAS

基于锁的同步方式，也是一种阻塞的线程间同步方式，无论是使用信号量，重入锁或者内部锁，受到核心资源的限制，不同线程间在锁竞争时，总不能避免相互等待，从而阻塞当前线程。为了避免这个问题，非阻塞同步的方式就被提出，最简单的一种非阻塞同步就以 ThreadLocal 为代表，每个线程拥有各自独立的变量副本，因此在并行计算时，无需相互等待。

本节将介绍一种更为重要的，基于比较并交换（Compare And Swap）CAS 算法的无锁并发控制方法。

与锁的实现相比，无锁算法的设计和实现都要复杂得多，但由于其非阻塞性，它对死锁问题天生免疫，并且，线程间的相互影响也远远比基于锁的方式要小。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此，它要比基于锁的方式拥有更优越的性能。

CAS 算法的过程是这样：它包含 3 个参数 CAS(V,E,N)。V 表示要更新的变量，E 表示预期值，N 表示新值。仅当 V 值等于 E 值时，才会将 V 的值设为 N，如果 V 值和 E 值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS 返回当前 V 的真实值。CAS 操作是抱着乐观的态度进行的，它总是认为自己可以成功完成操作。当多个线程同时使用 CAS 操作一个变量时，只有一个会胜出并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，CAS 操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

在硬件层面，大部分的现代处理器都已经支持原子化的 CAS 指令。在 JDK 5.0 以后，虚拟机便可以使用这个指令来实现并发操作和并发数据结构。并且，这种操作在虚拟机中可以说是

无处不在。读者应该记得轻量级锁中展示的代码片段吧：

```
if (mark == (markOop) Atomic::cmpxchg_ptr(lock, obj()->mark_addr(), mark)) {
```

上面的代码正是使用了 CAS 操作将锁地址复制到对象头的 Mark Word 中。

8.4.2 原子操作

为了能让 CAS 操作被 Java 应用程序充分使用，在 JDK 的 java.util.concurrent.atomic 包下，有一组使用无锁算法实现的原子操作类，主要有 AtomicInteger、AtomicIntegerArray、AtomicLong、AtomicLongArray 和 AtomicReference 等。它们分别封装了对整数、整数数组、长整型、长整型数组和普通对象的多线程安全操作。

以 AtomicInteger 为例，它的核心方法有：

```
public final int get()                                  //取得当前值
public final void set(int newValue)                     //设置当前值
public final int getAndSet(int newValue)                //设置新值，并返回旧值
public final boolean compareAndSet(int expect, int u)
                                              //如果当前值为 expect，则设置为 u
public final int getAndIncrement()                      //当前值加 1，返回旧值
public final int getAndDecrement()                      //当前值减 1，返回旧值
public final int getAndAdd(int delta)                   //当前值增加 delta，返回旧值
public final int incrementAndGet()                      //当前值加 1，返回新值
public final int decrementAndGet()                      //当前值减 1，返回新值
public final int addAndGet(int delta)                   //当前值增加 delta，返回新值
```

以 getAndSet()方法为例，看一下 CAS 算法是如何工作的：

```
public final int getAndSet(int newValue) {
    for (;;) {
        int current = get();
        if (compareAndSet(current, newValue))
            return current;
    }
}
```

在 CAS 算法中，首先是一个无穷循环，在这里，这个无穷循环用于多线程间的冲突处理，即在当前线程受其他线程影响而更新失败时，会不停地尝试，直到成功。

方法 get()用于取得当前值，并使用 compareAndSet()方法进行更新，如果未受其他线程影响，则预期值就等于 current。因此，可以将值更新为 newValue，若更新成功，则退出循环。

如果受其他线程影响，则在第 4 行 compareAndSet()时，预期值就不等于 current，更新失败，则进行下一次循环，尝试继续更新，直到成功。

因此，在整个更新过程中，无需加锁，无需等待。从这段代码中也可以看到，无锁的操作实际上将多线程并发的冲突处理交由应用层自行解决，这不仅提升了系统性能，还增加了系统的灵活性。但相对的，算法及编码的复杂度也明显地增加了。

【示例 8-6】在 java.util.concurrent.atomic 包中的类的性能是非常优越的，现在以 AtomicInteger 为例，了解一下普通的同步方法和它们的性能差距。

```
public class Atomic {
   private static final int MAX_THREADS = 3;                //线程数
   private static final int TASK_COUNT = 3;                 //任务数
   private static final int TARGET_COUNT = 10000000;        //目标总数

   private AtomicLong acount =new AtomicLong(0L);           //无锁的原子操作
   private long count=0;

   static CountDownLatch cdlsync=new CountDownLatch(TASK_COUNT);
   static CountDownLatch cdlatomic=new CountDownLatch(TASK_COUNT);

   protected synchronized long inc(){                       //有锁的加法
      return ++count;
   }

   protected synchronized long getCount(){                  //有锁的操作
      return count;
   }

   public void clearCount(){
      count=0;
   }

   public class SyncThread implements Runnable{
      protected String name;
      protected long starttime;
      AtomicLess out;
      public SyncThread(AtomicLess o,long starttime){
         out=o;
         this.starttime=starttime;
      }
```

```
        @Override
        public void run() {
            long v=out.getCount();
            while(v<TARGET_COUNT){
                v=out.inc();
            }
            long endtime=System.currentTimeMillis();
            System.out.println("SyncThread spend:"+(endtime-starttime)+"ms"+" v="+v);
            cdlsync.countDown();
        }
    }

    public void testSync() throws InterruptedException{
        ExecutorService exe=Executors.newFixedThreadPool(MAX_THREADS);
        long starttime=System.currentTimeMillis();
        SyncThread sync=new SyncThread(this,starttime);
        for(int i=0;i<TASK_COUNT;i++){
            exe.submit(sync);
        }
        cdlsync.await();
        exe.shutdown();
    }
    public class AtomicThread implements Runnable{
        protected String name;
        protected long starttime;
        public AtomicThread(long starttime){
            this.starttime=starttime;
        }
        @Override
        public void run() {
            long v=acount.get();
            while(v<TARGET_COUNT){
                v=acount.incrementAndGet();
            }
            long endtime=System.currentTimeMillis();
            System.out.println("AtomicThread spend:"+(endtime-starttime)+"ms"+" v="+v);
            cdlatomic.countDown();
        }
    }

    public void testAtomic() throws InterruptedException{
```

```
73         ExecutorService exe=Executors.newFixedThreadPool(MAX_THREADS);
74         long starttime=System.currentTimeMillis();
75         AtomicThread atomic=new AtomicThread(starttime);
76         for(int i=0;i<TASK_COUNT;i++){
77             exe.submit(atomic);
78         }
79         cdlatomic.await();
80         exe.shutdown();
81     }
82
83     public static void main(String args[]) throws InterruptedException{
84         AtomicLess a=new AtomicLess();
85         a.testSync();
86         a.testAtomic();
87     }
88 }
```

上述代码测试了使用有锁的方式和无锁的 CAS 对同一个数字进行技术操作。根据程序中的设置，两种方式都使用 3 个线程并行技术。有锁的操作将计数值保存在 count 变量中，无锁的原子将计数值保存在 acount 中。代码第 24~53 行为有锁的同步操作对计数器的累加逻辑。代码第 54~81 行为原子操作的累加逻辑。

当数值累计到 TARGET_COUNT（10000000）时，程序停止，并输出线程的工作时间，以便比较两者的性能。在笔者的计算机上，运行上述程序，得到的结果如下所示：

```
SyncThread spend:1752ms v=10000000
SyncThread spend:1752ms v=10000002
SyncThread spend:1752ms v=10000001
AtomicThread spend:718ms v=10000002
AtomicThread spend:718ms v=10000000
AtomicThread spend:718ms v=10000001
```

可以看到，使用原子操作对计数器进行累加时，性能远远高于传统的锁操作（锁操作耗时约 1752ms，而原子操作耗时约 718ms）。这也是为什么 CAS 指令在虚拟机内部被大量使用的原因。

8.4.3　新宠儿 LongAddr

前文中已经提到，无锁的原子类操作使用系统的 CAS 指令，有着远远超越锁的性能。那是否有可能在性能上更上一层楼呢？答案是肯定的。在 JDK 1.8 中引入了 LongAdder 类，这个类

也在 java.util.concurrent.atomic 包下，因此，可以推测，它也是使用了 CAS 指令。

前文中已经介绍了 AtomicInteger 等原子类的实现机制，它们都是在一个死循环内，不断尝试修改目标值，直到修改成功。如果竞争不激烈，那么修改成功的概率就很高，否则，修改失败的概率就很高，在大量修改失败时，这些原子操作就会进行多次循环尝试，因此性能就会受到影响。

结合前文介绍的减小锁粒度与 ConcurrentHashMap 的实现，读者应该可以想到一种对传统 AtomicInteger 等原子类的改进思路。虽然在 CAS 操作中没有锁，但是像减小锁粒度这种分离热点的思想依然可以使用。一种可行的方案就是仿造 ConcurrentHashMap，将热点数据分离。比如，可以将 AtomicInteger 的内部核心数据 value 分离成一个数组，每个线程访问时，通过哈希等算法映射到其中一个数字进行计数，而最终的计数结果，则为这个数组的求和累加，如图 8.5 所示，显示了这种优化思路。其中，热点数据 value 被分离成多个单元 cell，每个 cell 独自维护内部的值，当前对象的实际值由所有的 cell 累计合成，这样，热点就进行了有效的分离，提高了并行度。LongAdder 正是使用了这种思想。

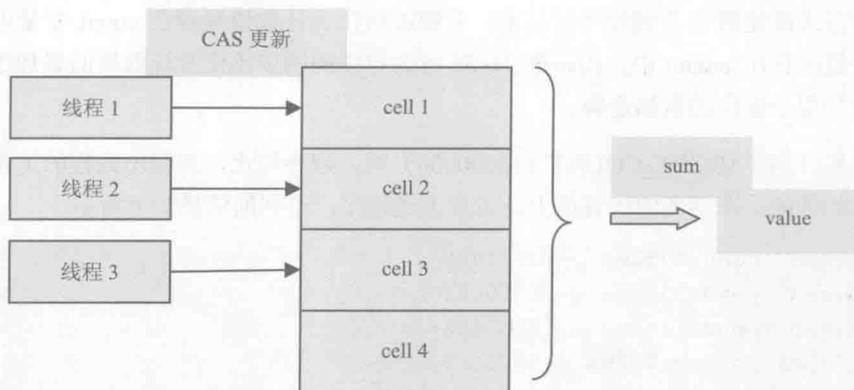


图 8.5　原子类的优化思路

【示例 8-7】继续上一节的演示代码，本节使用 LongAddr 进行累加计数，代码如下：

```
    private LongAdder lacount=new LongAdder();
 static CountDownLatch cdladdr=new CountDownLatch(TASK_COUNT);

 public class LongAddrThread implements Runnable{
        protected String name;
        protected long starttime;
        public LongAddrThread(long starttime){
            this.starttime=starttime;
```

```
            }
            @Override
            public void run() {
                long v=lacount.sum();
                while(v<TARGET_COUNT){
                    lacount.increment();
                    v=lacount.sum();
                }
                long endtime=System.currentTimeMillis();
                System.out.println("LongAdder spend:"+(endtime-starttime)+"ms"+
" v="+v);
                cdladdr.countDown();
            }
        }

        public void testAtomicLong() throws InterruptedException{
            ExecutorService exe=Executors.newFixedThreadPool(MAX_THREADS);
            long starttime=System.currentTimeMillis();
            LongAddrThread atomic=new LongAddrThread(starttime);
            for(int i=0;i<TASK_COUNT;i++){
                exe.submit(atomic);
            }
            cdladdr.await();
            exe.shutdown();
        }
```

执行这段代码，并和普通的原子操作、锁操作进行比较，输出结果如下：

```
SyncThread spend:1647ms v=10000001
SyncThread spend:1647ms v=10000002
SyncThread spend:1647ms v=10000000
AtomicThread spend:718ms v=10000000
AtomicThread spend:718ms v=10000002
AtomicThread spend:718ms v=10000001
LongAdder spend:211ms v=10000000
LongAdder spend:211ms v=10000000
LongAdder spend:211ms v=10000000
```

可以看到，就计数性能而言，LongAdder 已经远超普通的原子操作了。其中，锁操作耗时约 1647ms，原子操作耗时约 718ms，而 LongAddr 仅需要 211ms 左右。

8.5 将随机变为可控：理解 Java 内存模型

与串行程序相比，并发程序还必须额外处理一个问题，那就是多线程间数据的访问一致性。对于串行程序来说，如果线程修改了变量 A，那在修改之后的任意时间，读取变量 A 的值必定是修改后的新值。但对于并发程序，如果在一个线程中修改了全局变量 A，在另外一个线程中一定可以读取这个新值吗？答案是：不一定！但一旦出现多个线程访问某个变量的值不一致的情况，系统就有可能发生一些莫名其妙的问题，因此，在进行多线程程序设计时，必须要考虑这种情况。Java 内存模型（JMM）就是用来解释并规范这种情况的，将这种看似随机的状态变为可控，从而屏蔽多线程间可能引发的种种问题。本节将主要介绍 Java 内存模型的含义，以及它的基本原则和特性。

8.5.1 原子性

原子性中的原子代表不可分割的意思。原子操作是不可中断的，也不能被多线程干扰。比如，对 int 和 byte 等数据的赋值操作就具备基本的原子特性，而像“a++”这样的操作不具备原子性，因为它涉及读取 a、计算新值和写入 a 三步操作，中间有可能被其他线程干扰，导致最终的计算结果和实际值出现偏差。

在 32 位 Java 虚拟机系统中，对于 long 和 double 的赋值读取，由于 long 和 double 长度为 64 位，无法一次性操作，因此对于它们的操作都不是原子的，在并发环境下，可能会出现一些意想不到的错误。

【示例 8-8】下面的代码显示了对 long 型数据的并发写和读。

```
public class MultiThreadVolatileLong {
    public static long t=0;
    public static class ChangeT implements Runnable{
        private long to;
        public ChangeT(long to){
            this.to=to;
        }
        @Override
        public void run() {
            while(true){
            MultiThreadVolatileLong.t=to;
            Thread.yield();
            }
        }
```

```
15     }
16     public static class ReadT implements Runnable{
17         @Override
18         public void run() {
19             while(true){
20              long tmp=MultiThreadVolatileLong.t;
21              if(tmp!=111L && tmp!=-999L && tmp!=333L && tmp!=-444L)
22                  System.out.println(tmp);
23             Thread.yield();
24             }
25         }
26     }
27 
28     public static void main(String[] args) {
29         new Thread(new ChangeT(111L)).start();
30         new Thread(new ChangeT(-999L)).start();
31         new Thread(new ChangeT(333L)).start();
32         new Thread(new ChangeT(-444L)).start();
33         new Thread(new ReadT()).start();
34     }
35 }
```

上述代码使用 4 个线程对变量 t 进行赋值，4 个线程分别对 t 的赋值为 111、-999、333 和-444。因此，正常情况下，在任意时刻，t 的取值一定是其中一个。在程序第 21 行，读取线程读取变量 t 的值，并判断是否是这 4 个值的其中一个，如果不是则输出这个值。笔者在 JDK 1.7 32 位虚拟机上执行上述代码，部分输出结果如下：

```
4294966297
-4294967185
4294966852
4294966852
```

可以看到，程序有了输出，并且不是 4 个值中的任意一个。为什么会发生这种奇怪的现象呢？这就是因为多线程对 long 型数据的读写并非原子操作，因此，有可能出现一个线程写了 long 型数据中的 32 位，而另外一个线程写了 long 型的另外 32 位，导致这种异常情况的发生。

为了避免这种情况，最简单的处理方法是将第 2 行的：

```
public static long t=0
```

改为：

```
public static volatile long t=0
```

使用 volatile 之后，多个线程间对变量 t 的写入就能确保基本的原子特性，使得变量 t 的值必定为上述这 4 个取值之一。

8.5.2 有序性

现代的 CPU 都支持指令流水线执行。为了保证流水线的顺畅执行，在指令执行时，有可能会对目标指令进行重排。重排不会导致单线程中的语义修改，但会导致多线程中的语义出现不一致。即，在一个线程中观察另外一个线程的操作，我们会发现，被观察线程的指令顺序和预期情况不符。

指令重排会保证线程内串行语义一致。比如，以下语句不能进行重排：

- 写后读：a=1;b=a;
- 写后写：a = 1;a = 2;
- 读后写：a = b;b = 1;

很显然，如果将以上语句进行重排，最终的结果即使在同一个线程中，也会与原始指令语义不符，比如将 a=1;b=a 重排为 b=a;a=1，那么 b 的值在重排前后就不再相同，语义上有明显差异。而类似如 a=1;b=2;之类的指令就可以进行重排。因为在同一个线程中，先对 a 赋值，还是先对 b 赋值并没有语义上的冲突，执行完成后，效果是等价的。但是从其他线程观察本线程时，就有可能观察到 b 先赋值，然后再是 a 赋值。因此，在其他线程中通过 b 的值来推断 a 的值是不安全的（因为两者谁先赋值是不确定的）。

【示例 8-9】下例展示了由于指令重排引起的多线程间的语义冲突。

```
class OrderExample {
int a = 0;
boolean flag = false;
public void writer() {
    a = 1;
    flag = true;
}
public void reader() {
    if (flag) {
        int i =  a +1;
        ……
    }
}
}
```

假设线程 A 首先执行 writer()方法，接着线程 B 执行 reader()方法，如果发生指令重排，那么线程 B 在代码第 10 行时，不一定能看到 a 已经被赋值为 1 了。如图 8.6 所示，显示了两个线程的调用关系。

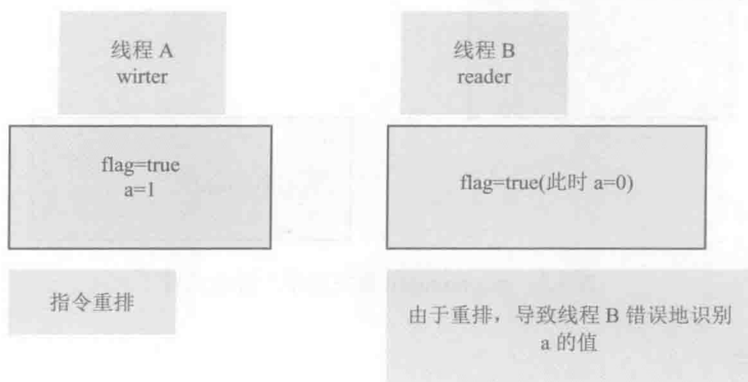


图 8.6　指令重排引起线程间语义不一致

解决有序性的一个简单方法就是使用 synchronized 关键字，如下所示：

```
01 class OrderExample {
02 int a = 0;
03 boolean flag = false;
04
05 public synchronized void writer() {
06     a = 1;
07     flag = true;
08 }
09
10 public synchronized void reader() {
11     if (flag) {
12         int i =  a +1;
13         ......
14     }
15 }
16 }
```

当使用 synchronized 后，由于同步，可以解决这种语义上的冲突，如图 8.7 所示，即使线程 A 进行了指令重排，但在 writer()方法执行时，线程 B 无法进入，只有线程 A 释放锁，线程 B 才得以进入，因此，无论线程 A 的指令执行顺序如何，线程 B 都会看到相同的最终结果。

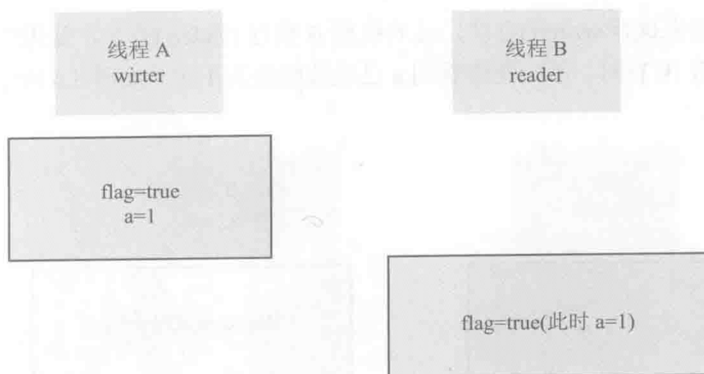


图 8.7　synchronized 解决指令重排的问题

8.5.3　可见性

可见性是指当一个线程修改了一个变量的值，在另外一个线程中可以马上得知这个修改。上一节介绍的指令重排就有可能使得一个线程可能无法立即得知一个变量的修改。此外，由于系统编译器优化，部分变量的值可能会被寄存器或者高速缓冲（Cache）缓存，而每个 CPU 都拥有独立的寄存器和 Cache，从而导致其他线程无法立即发现这个修改。

【示例 8-10】下面的代码显示了多线程间的可行性问题。

```
public class VolatileTest {
   public static class MyThread extends Thread{
      private boolean stop = false;
      public void stopMe(){
         stop=true;
      }

      public void run() {
         int i = 0;
         while (!stop) {
            i++;
         }
         System.out.println("Stop Thread");
      }
   }

   public static void main(String[] args) throws InterruptedException {
      MyThread t = new MyThread();
```

```
19         t.start();
20         Thread.sleep(1000);
21         t.stopMe();
22         Thread.sleep(1000);
23
24     }
25 }
```

上述代码开启两个线程，主线程和 MyThread 线程，在主线程中使用 stopMe()方法修改 stop 变量通知 MyThread 线程结束（代码第 21 行）。使用-server 参数执行这段代码（由于 server 虚拟机会做足够多的优化，可将多线程的可见性问题表现得更明显），结果发现，MyThread 始终无法结束。这就是由于在主线程中对 stop 变量的修改无法反应到 MyThread 线程中去。

解决可见性问题最简单的方法是将第 3 行代码修改为（增加 volatile 关键字）：

```
private volatile boolean stop = false;
```

在使用 volatile 之后，再次使用-server 参数执行这段代码，MyThread 线程就可以及时发现 stop 变量的变化，将线程退出。

除了 volatile 外，使用 synchronized 关键字也可以解决可见性问题。如下代码改写了 MyThread 线程，使用 synchronized 关键字解决线程间可见性问题。

```
public static class MyThread extends Thread{
    private boolean stop = false;
    public synchronized void stopMe(){
        stop=true;
    }
    public synchronized boolean stopped(){
        return stop;
    }
    public void run() {
        int i = 0;
        while (!stopped()) {
            i++;
        }
        System.out.println("Stop Thread");
    }
}
```

在使用同步方法后，MyThread 也可以正常接收到停止命令，将线程退出。由此，可以看到 synchronized 关键字不仅可以用于线程同步控制，还可以用于解决可见性问题。

8.5.4 Happens-Before 原则

在前文已经介绍了指令重排，虽然虚拟机和执行系统会对指令进行一定的重排，但是指令重排是有原则的，并非所有的指令都可以随便改变执行位置。以下罗列了一些基本原则，这些原则是指令重排不可违背的。

- 程序顺序原则：一个线程内保证语义的串行性。
- volatile 规则：volatile 变量的写，先发生于读，这保证了 volatile 变量的可见性。
- 锁规则：解锁（unlock）必然发生在随后的加锁（lock）前。
- 传递性：A 先于 B，B 先于 C，那么 A 必然先于 C。
- 线程的 start()方法先于它的每一个动作。
- 线程的所有操作先于线程的终结（Thread.join()）。
- 线程的中断（interrupt()）先于被中断线程的代码。
- 对象的构造函数执行结束先于 finalize()方法。

8.6 小结

本章主要介绍一些有关“锁”在虚拟机中的实现和优化，主要包括偏向锁、轻量级锁、自旋锁和锁膨胀。此外，本章还提供了一些在应用层面对多线程锁优化的一般思路，包括减少锁持有时间、减少锁粒度、锁分离、锁粗化等。对于锁的优化，无论是虚拟机层面还是应用层面，都需要进行权衡和深思，因为它们无法对所有的情况都取得较好的效果。此外，本章还介绍了无锁（CAS）这种高效的并行方式。最后，简要介绍了 Java 的内存模型。

第 9 章 Class 文件结构

对于 Java 虚拟机来说，Class 文件是虚拟机的一个重要接口。无论使用何种语言进行软件开发，只要能将源文件编译为正确的 Class 文件，那么这种语言就可以在 Java 虚拟机上执行。可以说，Class 文件就是 Java 虚拟机的基石。本章将详细介绍 Class 文件的组成，帮助大家更进一步了解虚拟机的运行过程。

本章涉及的主要知识点有：

- Class 文件的基本结构。
- 使用 jClassLib 查看 Class 文件。
- 使用 ASM 手工生成 Class 文件。

9.1 不仅跨平台，还能跨语言：语言无关性

Java 虚拟机提供了 Java 语言的跨平台功能。使用不同平台的 Java 虚拟机，可以让同一份

Class 文件运行在不同的平台上。这是 Java 的巨大优势。但就目前的软件水平来说，这个优势不再那么吸引人了，因为许许多多的语言都已经具备了这个特点。比如 Python、PHP、Perl、Ruby、Lisp 这些脚本语言，依靠其强大的解释器，天生就和执行平台无关。即便是相对封闭的.NET 体系，依然可以依靠 Mono 运行于 Linux 平台之上。跨平台似乎已经快成为一门语言必选的特性。

本书并不打算讨论 Java 语言，而是会花更多的精力讨论 Java 虚拟机平台。作为一个平台，Java 虚拟机不仅提供了跨平台功能，甚至还提供了跨语言的特性。从理论上说，无论使用何种语言编写你的软件，都可以让它在任意平台上执行。这看上去是一个何等诱人的特性啊！而实现这一切的基础，就是统一而强大的 Class 文件结构，它是异构语言和 Java 虚拟机之间的重要桥梁。

目前，微软.NET 平台是跨语言的典范，无论是 C#或者 VB 都可以在.NET 平台中执行，而随着 Java 虚拟机的不断发展，在这个方面，依靠着强大的社区力量，大有赶超.NET 平台的趋势。当下，诸如 Clojure（Lisp 语言的一种方言）、Groovy、Scala、Jython（Python）等语言都活跃在 Java 虚拟机平台之上。

图 9.1 显示了各种语言由源代码被编译成 Class 文件，并最终得以在 Java 虚拟机上执行的过程。

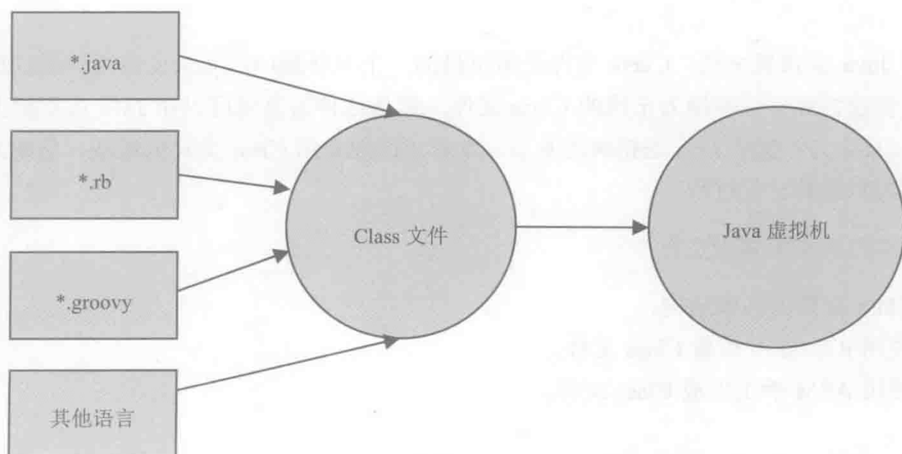


图 9.1　各种语言在 Java 虚拟机上执行

9.2 虚拟机的基石：Class 文件

分析 Class 文件的基本结构是本章最为主要的内容。Class 文件的结构并不是一成不变的，随着 Java 虚拟机的不断发展，总是不可避免地会对 Class 文件结构做出一些调整，但是其基本结构和框架是非常稳定的。Class 文件的总体结构如图 9.2 所示。

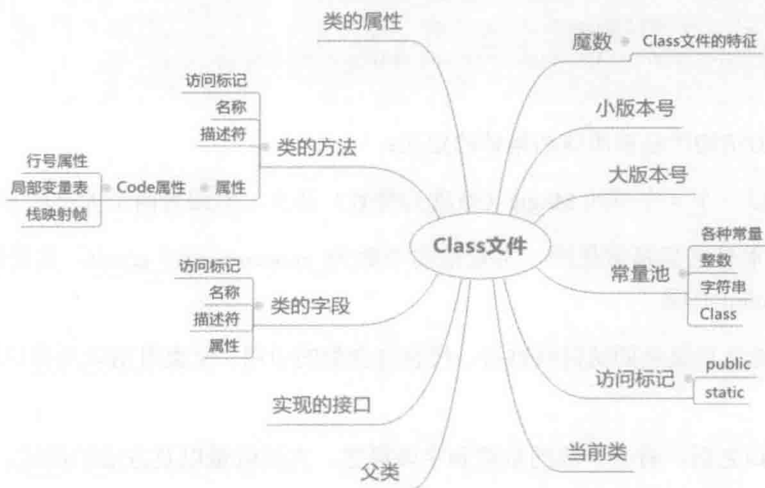


图 9.2 Class 文件总体结构

在了解总体结构之后，本节将使用比较严谨的方式，详细说明 Class 文件的各个组成。首先，简要说明一下用于描述 Class 文件结构的基本数据类型。

在 Java 虚拟机规范中，Class 文件使用一种类似于 C 语言结构体的方式进行描述，并且统一使用无符号整数作为其基本数据类型，由 u1、u2、u4、u8 分别表示无符号单字节、2 字节、4 字节和八字节整数。对于字符串，则使用 u1 数组进行表示。

根据 Java 虚拟机规范的定义，一个 Class 文件可以非常严谨地被描述成：

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info          constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
```

```
    u2              super_class;
    u2              interfaces_count;
    u2              interfaces[interfaces_count];
    u2              fields_count;
    field_info        fields[fields_count];
    u2              methods_count;
    method_info      methods[methods_count];
    u2              attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class 文件的结构严格按照该结构体的定义：

（1）文件以一个 4 字节的 Magic（被称为魔数）开头，紧跟着两个大小版本号。

（2）在版本号之后是常量池，常量池的个数为 constant_pool_count，常量池中的表项有 constant_pool_count-1 项。

（3）常量池之后是类的访问修饰符、代表自身类的引用、父类引用以及接口数量和实现的接口引用。

（4）在接口之后，有着字段的数量和字段描述、方法数量以及方法的描述。

（5）最后，存放着类文件的属性信息。

本节接下来将详细分析各个字段的含义。

9.2.1 Class 文件的标志——魔数

魔数（Magic Number）作为 Class 文件的标志，用来告诉 Java 虚拟机，这是一个 Class 文件。魔数是一个 4 个字节的无符号整数，它固定为 0xCAFEBABE。众所周知，Java 的名字和咖啡有着不解之缘。当你看到 CAFEBABE 时，会想到什么？像不像 Café Baby（咖啡宝贝）呢？因此，Java 的开发者们就用了这个整数来表示 Class 文件。

【示例 9-1】如果一个 Class 文件不以 0xCAFEBABE 开头，虚拟机在进行文件校验的时候就会直接抛出以下错误：

```
  Exception in thread "main" java.lang.ClassFormatError: Incompatible magic
value 184466110 in class file hsdb/Test2
      at java.lang.ClassLoader.defineClass1(Native Method)
      at java.lang.ClassLoader.defineClass(ClassLoader.java:791)
      at
```

```
java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
      at java.net.URLClassLoader.defineClass(URLClassLoader.java:449)
      at java.net.URLClassLoader.access$100(URLClassLoader.java:71)
      at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
      at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
      at java.security.AccessController.doPrivileged(Native Method)
      at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
      at java.lang.ClassLoader.loadClass(ClassLoader.java:423)
      at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
      at java.lang.ClassLoader.loadClass(ClassLoader.java:356)
      at hsdb.Test.<clinit>(Test.java:4)
      at hsdb.Main.main(Main.java:5)
```

注意： 魔数表示 Class 文件的标示符，在程序开发时，软件设计者喜欢使用一些特殊的数字来表示固定的文件类型或者特殊的含义。除了 Java 的 Class 文件以外，常用的 TAR 文件、PE 文件，甚至是网络 DHCP 报文内部，都会有类似的设计手法。

【示例 9-2】下面以一段简单的代码来展示魔数的内容。

```
package geym.zbase.ch9;

public class SimpleUser {
   public static final int TYPE = 1;

   private int id;
   private String name;

   public int getId() {
      return id;
   }

   public void setId(int id) throws IllegalStateException {
      try {
         this.id = id;
      } catch (IllegalStateException e) {
         System.out.println(e.toString());
      }
   }

   public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

提示：请读者留意以上代码，在后面的 Class 文件分析中，依然会以这段代码为例。

使用软件 WinHex 打开以上代码生成的 Class 文件，可以很容易看到魔数，如图 9.3 所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	CA	FE	BA	BE	00	00	00	33	00	37	07	00	02	01	00	19
00000010	67	65	79	6D	2F	7A	62	61	73	65	2F	63	68	39	2F	53
00000020	69	6D	70	6C	65	55	73	65	72	07	00	04	01	00	10	6A
00000030	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01
00000040	00	04	54	59	50	45	01	00	01	49	01	00	0D	43	6F	6E

图 9.3　Class 文件的魔数

9.2.2　Class 文件的版本

在魔数后面，紧跟着 Class 的小版本和大版本号。这表示当前 Class 文件，是由哪个版本的编译器编译产生的。首先出现的是小版本号，是一个两个字节的无符号整数，在此之后为大版本号，也用两个字节表示。

版本号和 Java 编译器的对应关系如表 9.1 所示。

表 9.1　Class文件版本号和平台的对应

大版本（十进制）	小　版　本	编译器版本
45	3	1.1
46	0	1.2
47	0	1.3
48	0	1.4
49	0	1.5
50	0	1.6
51	0	1.7
52	0	1.8

【示例 9-3】如图 9.4 所示，大版本号为 0x33，换算为 16 进制为 51，因此可以判断该 Class 文件是由 JDK 1.7 的编译器，或者在 JDK 1.8 下，使用“-target 1.7”参数编译生成的。

注意：0x33 是 16 进制表示，本书中如无特殊说明，16 进制数字前会加上 0x 前缀，以区别 10 进制数字。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	CA	FE	BA	BE	00	00	00	33	00	37	07	00	02	01	00	19
00000010	67	65	79	6D	2F	7A	62	61	73	65	2F	63	68	39	2F	53
00000020	69	6D	70	6C	65	55	73	65	72	07	00	04	01	00	10	6A
00000030	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01
00000040	00	04	54	59	50	45	01	00	01	49	01	00	0D	43	6F	6E

图 9.4　Class 文件的版本号

目前，高版本的 Java 虚拟机可以执行由低版本编译器生成的 Class 文件，但是低版本的 Java 虚拟机不能执行由高版本编译器生成的 Class 文件。

【示例 9-4】以下错误就是由于使用 1.7 的虚拟机，试图执行由 1.8 编译器产生的 Class 文件所导致的，注意加粗部分的错误描述。

```
    Exception in thread "main" java.lang.UnsupportedClassVersionError: hsdb/
Main: Unsupported major.minor version 52.0
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:791)
         省略部分输出
        at java.lang.ClassLoader.loadClass(ClassLoader.java:423)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:356)
        at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:482)
```

在实际应用中，由于开发环境和生产环境的不同，可能会导致该问题的发生。因此，需要我们在开发时，特别注意开发编译的 JDK 版本和生产环境中的 JDK 版本是否一致。

9.2.3　存放所有常数——常量池

常量池是 Class 文件中内容最为丰富的区域之一。随着 Java 虚拟机的不断发展，常量池的内容也日渐丰富。同时，常量池对于 Class 文件中的字段和方法解析也有着至关重要的作用，可以说，常量池是整个 Class 文件的基石。在版本号之后，紧跟着的是常量池的数量，以及若干个常量池表项。

【示例 9-5】如图 9.5 所示，0x37 表示该 Class 文件中合计有常量池表项 55-1=54 项（常量池 0 为空缺项，不存放实际内容，0x37 换算了 10 进制为 55）。在数量之后，就是常量池的实际内容，每一项以类型、长度、内容或者类型、内容的格式存放依次排列。图 9.5 限于篇幅有

限，并未显示该 Class 文件的所有常量。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000  CA FE BA BE 00 00 00 33  00 37 07 00 02 01 00 19  Êþº¾...3.7......
00000010  67 65 79 6D 2F 7A 62 61  73 65 2F 63 68 39 2F 53  geym/zbase/ch9/S
00000020  69 6D 70 6C 65 55 73 65  72 07 00 04 01 00 10 6A  impleUser......j
00000030  61 76 61 2F 6C 61 6E 67  2F 4F 62 6A 65 63 74 01  ava/lang/Object.
00000040  00 04 54 59 50 45 01 00  01 49 01 00 0D 43 6F 6E  ..TYPE...I...Con
00000050  73 74 61 6E 74 56 61 6C  75 65 03 00 00 00 01 01  stantValue......
00000060  00 02 69 64 01 00 04 6E  61 6D 65 01 00 12 4C 6A  ..id...name...Lj
00000070  61 76 61 2F 6C 61 6E 67  2F 53 74 72 69 6E 67 3B  ava/lang/String;
00000080  01 00 06 3C 69 6E 69 74  3E 01 00 03 28 29 56 01  ...<init>...()V.
00000090  00 04 43 6F 64 65 0A 00  03 00 10 0C 00 0C 00 0D  ..Code.........
```

图 9.5 常量池数量和表项

在详细解析这段常量池之前，先给出一个常量池表项的类型，参见表 9.2，它表示常量池可能出现的内容，其中 TAG 为表示该常量的整数枚举值。

表 9.2 常量池表项和其TAG值

常量池类型	TAG	常量池类型	TAG
CONSTANT_Class	7	CONSTANT_Fieldref	9
CONSTANT_Methodref	10	CONSTANT_InterfaceMethodref	11
CONSTANT_String	8	CONSTANT_Integer	3
CONSTANT_Float	4	CONSTANT_Long	5
CONSTANT_Double	6	CONSTANT_NameAndType	12
CONSTANT_Utf8	1	CONSTANT_MethodHandle	15
CONSTANT_MethodType	16	CONSTANT_InvokeDynamic	18

作为常量池底层的数据类型 CONSTANT_Utf8、CONSTANT_Integer、CONSTANT_Float、CONSTANT_Long、CONSTANT_Double 分别表示 UTF8 字符串、整数、浮点数、长整型和双精度浮点常量。

CONSTANT_Utf8 的格式如下定义：

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

根据表 9.2 可以看出，UTF8 的 tag 值为 1。紧接着，是该字符串的长度 length，最后是字符串的内容。

【示例 9-6】如图 9.6 所示，0x01 表示为一个 UTF8 的常量，接着 0x0019 表示该常量一共

25 个字节。因此，从 0x0019 之后数 25 个字节就为该常量的实际内容。从图中可知，该内容为字符串 geym/zbase/ch9/SimpleUser。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	CA	FE	BA	BE	00	00	00	33	00	37	07	00	02	01	00	19	Êþº¾...3.7......
00000010	67	65	79	6D	2F	7A	62	61	73	65	2F	63	68	39	2F	53	geym/zbase/ch9/S
00000020	69	6D	70	6C	65	55	73	65	72	07	00	04	01	00	10	6A	impleUser......j

图 9.6　UTF8 常量的表示

UTF8 的常量经常被其他类型的常量引用，比如在本例中，CONSTANT_Class 常量就会引用该 UTF8 作为类名。CONSTANT_Class 的结构如下：

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

其中 tag 为 7，表示一个 CONSTANT_Class 常量，第 2 个字段为一个两个字节的整数，表示常量池的索引，在 CONSTANT_Class 中，该索引指向的常量必须是 CONSTANT_Utf8。

【示例 9-7】如图 9.7 所示，0x07 表示该常量为 CONSTANT_Class，0x02 表示该类的类名由常量池第 2 个常量字符串指定。在本例中，该常量就是上文分析所得的 geym/zbase/ch9/SimpleUser。因此，该 CONSTANT_Class 表示的 Class 类型为 geym.zbase.ch9. SimpleUser。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	CA	FE	BA	BE	00	00	00	33	00	37	07	00	02	01	00	19	Êþº¾...3.7......
00000010	67	65	79	6D	2F	7A	62	61	73	65	2F	63	68	39	2F	53	geym/zbase/ch9/S
00000020	69	6D	70	6C	65	55	73	65	72	07	00	04	01	00	10	6A	impleUser......j

图 9.7　CONSTANT_Class 常量的表示

CONSTANT_Integer、CONSTANT_Float、CONSTANT_Long、CONSTANT_Double 分别表示数字的字面量。当使用 final 定义一个数字常量时，Class 文件中就会生成一个数字的常量。它们的结构分别为：

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}

CONSTANT_Float_info {
    u1 tag;
```

```
    u4 bytes;
}

CONSTANT_Long_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}

CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

其中，tag 的值依然参见表 9.2。对于 CONSTANT_Integer 和 CONSTANT_Float，它们的值由一个 4 字节的无符号整数表示，对于 CONSTANT_Long 和 CONSTANT_Double，它们的值由两个 4 字节无符号整数表示。这里以 CONSTANT_Integer 为例说明这些常量的表示方式。

【示例 9-8】如图 9.8 所示，0x03 表示一个整数常量，紧接着的 00 00 00 01 表示数字 1。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	CA	FE	BA	BE	00	00	00	33	00	37	07	00	02	01	00	19
00000010	67	65	79	6D	2F	7A	62	61	73	65	2F	63	68	39	2F	53
00000020	69	6D	70	6C	65	55	73	65	72	07	00	04	01	00	10	6A
00000030	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01
00000040	00	04	54	59	50	45	01	00	01	49	01	00	0D	43	6F	6E
00000050	73	74	61	6E	74	56	61	6C	75	65	03	00	00	00	01	01

图 9.8　CONSTANT_Integer 常量的表示

另外一个值得注意的字面量是 CONSTANT_String，它表示一个字符串常量，结构如下：

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}
```

其中 tag 为 8，一个 2 字节长的无符号整数指向常量池的索引，表示该字符串对应的 UTF8 内容。

另一个被广泛使用的类型为 CONSTANT_NameAndType，从名字上可以看出，它表示一个名词和类型，格式如下：

```
CONSTANT_NameAndType_info {
```

```
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

其中，tag 为 12，第一个 2 字节 name_index 表示名称，意为常量池的索引，表示常量池第 name_index 项为名字，通常可以表示字段名字或者方法名字。第二个 2 字节 descriptor_index 表示类型的描述，比如表示方法的签名或者字段的类型。

【示例 9-9】如图 9.9 所示，0x0C 表示一个 CONSTANT_NameAndType，0x0009 表示第 9 项常量为名称，查常量池，可得第 9 项常量为 id 字符串。0x0006 表示第 6 项常量，查常量表得到字符串 I，表示 int 类型。因此，该 CONSTANT_NameAndType 表示一个名称为 id，类型为 int 的表项。

```
000000F0  65 74 49 64 01 00 03 28  29 49 09 00 01 00 18 0C  etId...()I......
00000100  00 09 00 06 01 00 05 73  65 74 49 64 01 00 04 28  .......setId...(
00000110  49 29 56 01 00 0A 45 78  63 65 70 74 69 6F 6E 73  I)V...Exceptions
00000120  07 00 1D 01 00 1F 6A 61  76 61 2F 6C 61 6E 67 2F  ......java/lang/
```

图 9.9　CONSTANT_NameAndType 常量的表示

CONSTANT_NameAndType 的 descriptor_index 使用了一组特定的字符串来表示类型，如表 9.3 所示。

表 9.3　类型的字符串表示方法

字符串	类型	字符串	类型
B	byte	C	char
D	double	F	float
I	int	J	long
S	short	Z	boolean
V	void	L;	表示对象
[	数组		

对于对象类型来说，总是以 L 开头，紧跟类的全限定名，用分号（;）结尾，比如以字符串“Ljava/lang/Object;”表示类 java.lang.Object。数组则以左中括号“[”作为标记，比如 String 二维数组，使用“[[Ljava/lang/String;”字符串表示。

对于类的方法和字段，则分别使用 CONSTANT_Methodref 和 CONSTANT_Fieldref 表示。它们分别可以表示一个类的方法以及字段的引用。CONSTANT_Methodref 和 CONSTANT_Fieldref 的结构是非常类似的，如下所示：

```
CONSTANT_Methodref_info {
    u1 tag;
```

```
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

其中 CONSTANT_Methodref 的 tag 值为 10，CONSTANT_Fieldref 的 tag 值为 9。它们的 class_index 表示方法或者字段所在的类在常量池中的索引，它会指向一个 CONSTANT_Class 结构。第 2 项 name_and_type_index 也是指向常量池的索引，但表示一个 CONSTANT_NameAndType 结构，它定义了方法或者字段的名称、类型或者签名。

【示例 9-10】图 9.10 显示了本例中使用的 System.out.println()函数所表示的 CONSTANT_Methodref 在常量池中的引用关系。可以看到，Methodref 结构的 class index 字段指向了第 41 号常量池项，表示 Class，而该项又进一步指向常量池中的 UTF8 数据，表明该 Class 的类型。而 Methodref 的 name and type 字段则指向常量池第 43 项，NameAndType 类型的数据，它包括名字和类型两个字段，又分别指向常量池中的两个字符串 println 和(Ljava/lang/String;)V，表示方法的名字和方法的签名。(Ljava/lang/String;)V 表示该方法接收一个 String 类型的参数，并且返回值为 void。就这样，通过常量池中的引用关系，通过 Methodref 结构，将方法描述清楚了。

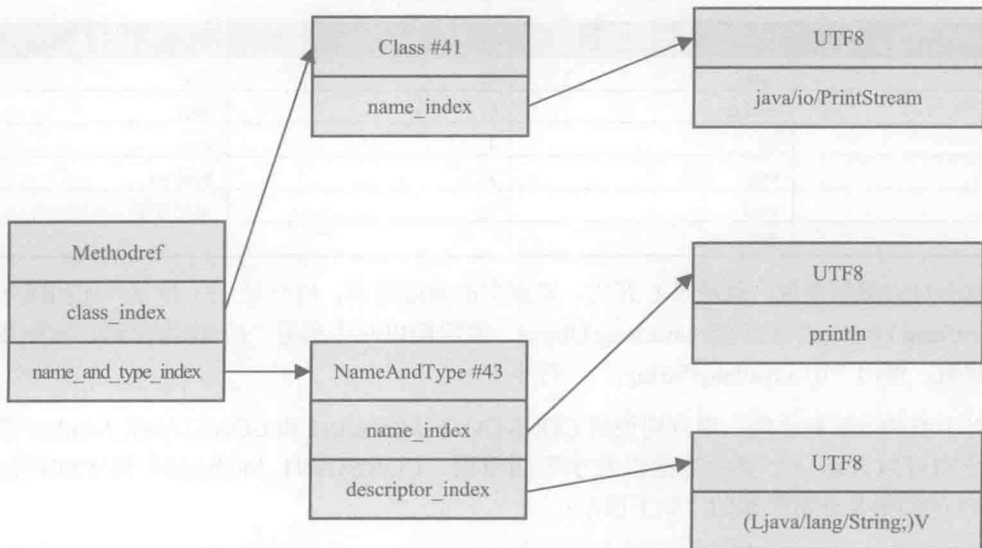


图 9.10　Methodref 常量池引用关系

【示例 9-11】图 9.11 显示了该 Methodref 常量的二进制表示。其中，0x0A 表示这是一个 CONSTANT_Methodref。0x0029 表示 Class 类型由常量池第 41 项指定。0x002B 表示该方法的名字和签名由常量池第 43 项决定。

```
000001A0  76 61 2F 6C 61 6E 67 2F  53 74 72 69 6E 67 3B 0A  va/lang/String;
000001B0  00 29 00 2B 07 00 2A 01  00 13 6A 61 76 61 2F 69   ) +  *   java/i
000001C0  6F 2F 50 72 69 6E 74 53  74 72 65 61 6D 0C 00 2C  o/PrintStream  ,
000001D0  00 2D 01 00 07 70 72 69  6E 74 6C 6E 01 00 15 28   -   println    (
```

图 9.11　Methodref 常量的二进制表示

字段 CONSTANT_Fieldref 和 CONSTANT_Methodref 是完全类似的，限于篇幅有限，不再赘述。

对于 CONSTANT_InterfaceMethodref，它用于表示一个接口的方法。如果在 Java 程序中，出现了对接口方法的调用，那么就会在常量池中生成一个接口方法的引用。该项目的结构如下：

```
CONSTANT_InterfaceMethodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

可以看到，它的使用方式和 CONSTANT_Methodref 是一样的。

最后，再来看一组常量类型 CONSTANT_MethodHandle、CONSTANT_MethodType 和 CONSTANT_InvokeDynamic。它们是在 JDK1.7 引入的新的常量类型，分别表示函数句柄、函数类型签名和动态调用。这里简单描述一下它们的含义和结构，具体的使用案例将在下一节中给出。

CONSTANT_MethodType 的结构定义如下：

```
CONSTANT_MethodType_info {
    u1 tag;
    u2 descriptor_index;
}
```

其中 tag 为 16，descriptor_index 为指向常量池的一个 UTF8 字符串的索引，使用手法和前面介绍的索引如出一辙。该常量项用于描述一个方法签名，比如“()V”，表示一个不接收参数，返回值为 void 的方法。当需要传送给引导方法一个 MethodType 类型时，类文件中就会出现此项。（有关引导方法和该类型具体使用案例，可参见第 11 章）

CONSTANT_MethodHandle 为一个方法句柄，它可以用来表示函数方法、类的字段或者构造函数等。方法句柄指向一个方法、字段，和 C 语言中的函数指针或者 C#中的委托有些类似。

它的结构如下：

```
CONSTANT_MethodHandle_info {
    u1 tag;
    u1 reference_kind;
    u2 reference_index;
}
```

其中，tag 值为 15，reference_kind 表示这个方法句柄的类型，reference_index 为指向常量池的索引，reference_index 具体指向的类型，由 reference_kind 确定。两者对应关系参见表 9.4。

表 9.4　MethodHandle字段含义

reference_kind取值	reference_index对应类型
REF_getField(1)	常量池的指向内容必须是CONSTANT_Fieldref类型
REF_getStatic(2)	
REF_putField(3)	
REF_putStatic(4)	
REF_invokeVirtual(5)	常量池指针必须是CONSTANT_Methodref类型，对于REF_invokeInterface来说为InterfaceMethodref类型，且不能为<init> 或者<clinit>方法（即不能为类的构造函数或者初始化方法）
REF_invokeStatic(6)	
REF_invokeSpecial(7)	
REF_invokeInterface(9)	
REF_newInvokeSpecial(8)	常量池指针必须是CONSTANT_Methodref类型，且对应的方法必须为<init>

CONSTANT_InvokeDynamic 结构用于描述一个动态调用，动态调用是 Java 虚拟机平台引入的，专门为动态语言提供函数动态调用绑定支持的功能。有关动态调用的详细举例，参见第 11 章。这里仅给出相关结构信息，如下所示：

```
CONSTANT_InvokeDynamic_info {
    u1 tag;
    u2 bootstrap_method_attr_index;
    u2 name_and_type_index;
}
```

其中，tag 为 18，bootstrap_method_attr_index 为指向引导方法表中的索引，即定位到一个引导方法。引导方法用于在动态调用时进行运行时函数查找和绑定。引导方法表属于类文件的属性（Attribute），name_and_type_index 为指向常量池的索引，且指向的表项必须是 CONSTANT_NameAndType，用于表示方法的名字以及签名。

9.2.4　Class 的访问标记（Access Flag）

在常量池后，紧跟着访问标记。该标记使用两个字节表示，用于表明该类的访问信息，如

public、final、abstract 等。

根据表 9.5 可以看出，每一种类型的表示都是通过设置访问标记的 32 位中的特定位来实现的。比如，若是 public final 的类，则该标记为 ACC_PUBLIC | ACC_FINAL。

表 9.5 类Access Flag标记位和含义

标记名称	数 值	描 述
ACC_PUBLIC	0x0001	表示public类（public类可以在包外访问）
ACC_FINAL	0x0010	是否为final类（final类不可被继承）
ACC_SUPER	0x0020	使用增强的方法调用父类方法
ACC_INTERFACE	0x0200	是否为接口
ACC_ABSTRACT	0x0400	是否是抽象类
ACC_SYNTHETIC	0x1000	由编译器产生的类，没有源码对应
ACC_ANNOTATION	0x2000	是否是注释
ACC_ENUM	0x4000	是否是枚举

【示例 9-12】以图 9.12 为例，该标记位为 0x0021，因此，可以判断该类为 public，且 ACC_SUPER 标记被置为 1。

提示：使用 ACC_SUPER 可以让类更准确地定位到父类的方法 super.method()，现代编译器都会设置并且使用这个标记。

```
00000260  55 73 65 72 2E 6A 61 76  61 00 21 00 01 00 03 00
00000270  00 00 03 00 19 00 05 00  06 00 01 00 07 00 00 00
```

图 9.12 访问标记的二进制表示

9.2.5 当前类、父类和接口

在访问标记后，会指定该类的类别、父类类别以及实现的接口，格式如下：

```
u2          this_class;
u2          super_class;
u2          interfaces_count;
u2          interfaces[interfaces_count];
```

其中，this_class、super_class 都是 2 字节无符号整数，它们指向常量池中一个 CONSTANT_Class，以表示当前的类型以及父类。由于在 Java 中只能使用单继承，因此，只需要保存单个父类即可。

注意：super_class 指向的父类不能是 final。

由于一个类可以实现多个接口，因此需要以数组形式保存多个接口的索引，表示接口的每个索引也是一个指向常量池的 CONSTANT_Class（当然这里就必须是接口，而不是类）。

如果该类没有实现任何接口，则 interfaces_count 为 0。

9.2.6 Class 文件的字段

在接口描述后，就会有类的字段信息。由于一个类会有多个字段，因此，需要首先指明字段的个数：

```
u2            fields_count;
field_info    fields[fields_count];
```

字段的数量 fields_count 是一个 2 字节无符号整数。字段数量之后为字段的具体信息，每一个字段为一个 field_info 的结构，该结构如下：

```
field_info {
   u2             access_flags;
   u2             name_index;
   u2             descriptor_index;
   u2             attributes_count;
   attribute_info attributes[attributes_count];
}
```

（1）首先是字段的访问标记，非常类似于类的访问标记，该字段的取值参见表 9.6。

表 9.6　字段Access Flag标记位和含义

标 记 名 称	数　值	描　述
ACC_PUBLIC	0x0001	表示public字段
ACC_PRIVATE	0x0002	表示private私有字段
ACC_PROTECTED	0x0004	表示protected保护字段
ACC_STATIC	0x0008	表示静态字段
ACC_FINAL	0x0010	是否为final字段，final字段表示常量
ACC_VOLATILE	0x0040	是否为volatile
ACC_TRANSIENT	0x0080	是否为瞬时字段，表示在持久化读写时，忽略该字段
ACC_SYNTHETIC	0x1000	由编译器产生的方法，没有源码对应
ACC_ENUM	0x4000	是否是枚举

（2）紧接着，是一个 2 字节整数，表示字段的名称，它指向常量池中的 CONSTANT_Utf8 结构。

（3）名称后的 descriptor_index 也指向常量池中 CONSTANT_Utf8，该字符用于描述字段

的类型，具体的表示方法参见表 9.3。

（4）一个字段还可能拥有一些属性，用于存储更多的额外信息，比如初始化值、一些注释信息等。属性个数存放在 attributes_count 中，属性具体内容存放于 attributes 数组。

以常量属性为例，常量属性的结构为：

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

常量属性的 attribute_name_index 为 2 字节整数，指向常量池的 CONSTANT_Utf8，并且这个字符串为“ConstantValue”。接着，为 attribute_length，它由 4 个字节组成，表示这个属性的剩余长度为多少。对常量属性而言，这个值恒为 2。最后的 constantvalue_index 表示属性值，但值并不直接出现在属性中，而是存放在常量池中，这里的 constantvalue_index 也是指向常量池的索引，并且存在如表 9.7 所示的对应关系。这表示，一个 int 类型字段的常量，constantvalue_index 指向的常量池类型必须是 CONSTANT_Integer。

表 9.7　常量数据类型和常量池类型对应关系

字 段 类 型	常量池表项类型
long	CONSTANT_Long
float	CONSTANT_Float
double	CONSTANT_Double
int, short, char, byte, boolean	CONSTANT_Integer
String	CONSTANT_String

【示例 9-13】以本章给出的 SimpleUser 为例，显示一个字段的完整结构，如图 9.13 所示。

首先 0x0003 表示该类 SimpleUser 存在 3 个字段。0x0019 为第一个字段的访问标记，查表 9.5，可以看到 0x0019=ACC_PUBLIC|ACC_STATIC|ACC_FINAL，因此这个字段是一个 public static final 的字段。接着，0x0005 为常量池索引，表示字段名称，查常量池表可得字符串“TYPE”，0x0006 为字段的类型描述，查常量池表得字符串“I”。由此，看到这是一个类型为 int，变量名为 TYPE 的 public static final 常量。接着为属性数量，值为 0x0001，表示该字段存在 1 个属性，0x0007 为属性名，通过该值确认属性的类型。查常量池第 7 项，为字符串“ConstantValue”，表示该属性为常量属性。之后，连续 4 个字节 0x00000002 为属性的剩余长度，这里表示从 0x00000002 之后的两个字节为属性的全部内容。本例中，该值为 0x0008，它表示属性值需要查阅常量池第 8 项。查找常量池第 8 项，得常量 CONSTANT_Integer，值为 1。所以该字段的常

量值为 1，类型为 int。



图 9.13　字段解析示例

9.2.7　Class 文件的方法基本结构

在字段之后，就是类的方法信息。方法信息和字段类似，由两部分组成：

```
u2             methods_count;
method_info     methods[methods_count];
```

其中 methods_count 为 2 字节整数，表示该类中有几个方法。接着就是 methods_count 个 method_info 结构，每一个 method_info 表示一个方法，该结构如下：

```
method_info {
    u2              access_flags;
    u2              name_index;
    u2              descriptor_index;
    u2              attributes_count;
    attribute_info attributes[attributes_count];
}
```

（1）其中，access_flag 为方法的访问标记，用于标明方法的权限以及相关特性，它的取值参见表 9.8。

表 9.8　方法访问标记取值

标记名称	值	作　用
ACC_PUBLIC	0x0001	public方法
ACC_PRIVATE	0x0002	private私有方法
ACC_PROTECTED	0x0004	protected方法
ACC_STATIC	0x0008	静态方法
ACC_FINAL	0x0010	final方法，不可被继承重载
ACC_SYNCHRONIZED	0x0020	synchronized同步方法
ACC_BRIDGE	0x0040	由编译器产生的桥接方法

续表

标记名称	值	作　用
ACC_VARARGS	0x0080	可变参数的方法
ACC_NATIVE	0x0100	native本地方法
ACC_ABSTRACT	0x0400	抽象方法
ACC_STRICT	0x0800	浮点模式为FP-strict
ACC_SYNTHETIC	0x1000	编译器产生的方法，没有源码对应

（2）在访问标记后，name_index 表示方法的名称，它是一个指向常量池的索引。descriptor_index 为方法描述符，它也是指向常量池的索引，是一个字符串，用以表示方法的签名（参数、返回值等），它基于表 9.3 所示的字符串的类型表示方法，同时对方法签名的表示做了一些规定。它将函数的参数类型写在一对小括号中，并在括号右侧给出方法的返回值。比如，若有如下方法：

```
Object m(int i, double d, Thread t) {..}
```

则它的方法描述符为：

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

可以看到，方法的参数统一列在一对小括号中，“I”表示 int，“D”表示 double，“Ljava/lang/Thread;”表示 Thread 对象。小括号右侧的 Ljava/lang/Object;表示方法的返回值为 Object 对象。

（3）和字段类似，方法也可以附带若干个属性，用于描述一些额外信息，比如方法字节码等，attributes_count 表示该方法中属性的数量，紧接着，就是 attributes_count 个属性的描述。

对于属性 attribute 来说，它们的统一格式为：

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

其中，attribute_name_index 表示当前 attribute 的名称，attribute_length 为当前 attribute 的剩余长度，紧接着就是 attribute_length 个字节的 byte 数组。

常用的 attribute 如表 9.9 所示。

表 9.9　常用属性Attribute

属性Attribute	作　　用
ConstantValue	用于字段常量
Code	表示方法的字节码
StackMapTable	Code属性的描述属性，用于字节码变量类型验证
Exceptions	方法的异常信息
SourceFile	类文件的属性，表示生成这个类的源码
LineNumberTable	Code属性的描述属性，描述行号和字节码的对应关系
LocalVariableTable	Code属性的描述属性，描述函数局部变量表
BootstrapMethods	类文件的描述属性，存放类的引导方法。用于invokeDynamic
StackMapTable	Code属性的描述属性，用于字节码类型校验

9.2.8　方法的执行主体——Code 属性

方法的主要内容存放在其属性之中，而当中最为重要的一个属性就是 Code。它存放着方法的字节码等信息，结构如下：

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {   u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Code 属性的第一个字段 attribute_name_index 指定了该属性的名称，它是一个指向常量池的索引，指向的类型为 CONSTANT_Utf8，对于 Code 属性来说，该值恒为“Code”。接着，attribute_length 指定了 Code 属性的长度，该长度不包括前 6 个字节，也就是剩余长度。

在方法执行过程中，操作数栈可能不停地变化，在整个执行过程中，操作数栈存在一个最大深度，该深度由 max_stack 表示。同理，在方法执行过程中，局部变量表也可能会不断变化。

在整个执行过程中局部变量表的最大值由 max_locals 表示，它们都是 2 字节的无符号整数，对于局部变量表可以进一步参考 2.4.1 节，有关操作数栈的使用，请参阅 11.1 节。

在 max_locals 之后，就是作为方法的最重要部分——字节码。它由 code_length 和 code[code_length]两部分组成，code_length 表示字节码的长度，为 4 字节无符号整数，code[code_length]为 byte 数组，为字节码内容本身。

在字节码之后，存放该方法的异常处理表。异常处理表告诉一个方法该如何处理字节码中可能抛出的异常。异常处理表亦由两部分组成：表项数量和内容。其中 exception_table_length 表示异常表的表项数量，exception_table[exception_table_length]结构为异常表。表中每一行由 4 部分组成，分别是 start_pc、end_pc、handler_pc 和 catch_type。这 4 项表示从方法字节码的 start_pc 偏移量开始到 end_pc 偏移量为止的这段代码中，如果遇到了 catch_type 所指定的异常，那么代码就跳转到 handler_pc 的位置执行。在这 4 项中，start_pc、end_pc 和 handler_pc 都是字节码的编译量，也就是在 code[code_length]中的位置，而 catch_type 为指向常量池的索引，它指向一个 CONSTANT_Class 类，表示需要处理的异常类型。

至此，Code 属性的主体部分已经介绍完毕，但是 Code 属性中还可能包含更多信息，比如行号、局部变量表等。这些信息都以 attribute 属性的形式内嵌在 Code 属性中，即除了字段、方法和类文件可以内嵌属性外，属性本身也可以内嵌其他属性。

9.2.9　记录行号——LineNumberTable 属性

Code 属性本身也包含着其他属性以进一步存储一些额外信息。首先，来看一下 LineNumberTable，它是 Code 属性的属性，用于描述 Code 属性。LineNumberTable 用来记录字节码偏移量和行号的对应关系，在软件调试时，该属性有着至关重要的作用，若没有它，则调试器无法定位到对应的源码。LineNumberTable 属性的结构如下：

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    {   u2 start_pc;
        u2 line_number;
    } line_number_table[line_number_table_length];
}
```

其中，attribute_name_index 为指向常量池的索引，在 LineNumberTable 属性中，该值为"LineNumberTable"，attribute_length 为 4 字节无符号整数，表示属性的长度（不含前 6 个字

节），line_number_table_length 表明了表项有多少条记录，line_number_table 为表的实际内容，它包含 line_number_table_length 个<start_pc，line_number>元组，其中，start_pc 为字节码偏移量，line_number 为对应的行号。

9.2.10 保存局部变量和参数——LocalVariableTable 属性

对 Code 属性而言，另外一个重要的属性是 LocalVariableTable，也就是局部变量表。它记录了一个方法中所有的局部变量，它的结构如下：

```
LocalVariableTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    {   u2 start_pc;
        u2 length;
        u2 name_index;
        u2 descriptor_index;
        u2 index;
    } local_variable_table[local_variable_table_length];
}
```

其中，attribute_name_index 为当前属性的名字，它是指向常量池的索引。对局部变量表而言，该值为“LocalVariableTable”，attribute_length 为属性的长度，local_variable_table_length 为局部变量表表项条目。

局部变量表的每一条记录由以下几个部分组成。

- start_pc、length：表示当前局部变量的开始位置（start_pc）和结束位置（start_pc+length，不含最后一个字节）。
- name_index：局部变量的名称，这是一个指向常量池的索引。
- descriptor_index：局部变量的类型描述，指向常量池的索引。使用和字段描述符一样的方式描述局部变量。
- index：局部变量在当前帧栈的局部变量表中的槽位。对于 long 和 double 的数据，它们会占据局部变量表中的两个槽位。

9.2.11 加快字节码校验——StackMapTable 属性

对于 JDK 1.6 以后的类文件，每个方法的 Code 属性还可能含有一个 StackMapTable 的属性结构。该结构中存有若干个叫做栈映射帧（stack map frame）的数据。该属性不包含运行时所需

的信息，仅用作 Class 文件的类型校验。

StackMapTable 的结构如下：

```
StackMapTable_attribute {
    u2             attribute_name_index;
    u4             attribute_length;
    u2             number_of_entries;
    stack_map_frame entries[number_of_entries];
}
```

其中，attribute_name_index 为常量池索引，恒为“StackMapTable”， attribute_length 为该属性的长度，number_of_entries 为栈映射帧的数量，最后的 entries 则为具体的内容，每一项为一个 stack_map_frame 结构。

每一个栈映射帧都是为了说明在一个特定的字节码偏移位置上，系统的数据类型是什么（包括局部变量表的类型和操作数栈的类型）。每一帧都会显式或者隐式地指定一个字节码偏移量的变化值 offset_delta，使用 offset_delta 可以计算出这一帧数据的字节码偏移位置。计算方法就是将 offset_delta+1 和上一帧的字节码偏移量相加。如果上一帧是方法的初始帧，那么，字节码偏移量为 offset_delta 本身。

注意： 这里说的“帧”，和帧栈的帧不是同一个概念。这里更接近于一个跳转语句，跳转语句将函数划分成不同的块，每一块的概念就接近于这里所说的栈映射帧中的“帧”。

StackMapTable 结构中的 stack_map_frame 被定义为一个枚举值，它可能的取值如下：

```
union stack_map_frame {
    same_frame;
    same_locals_1_stack_item_frame;
    same_locals_1_stack_item_frame_extended;
    chop_frame;
    same_frame_extended;
    append_frame;
    full_frame;
}
```

对于每一种可能的取值，都有自己特定的含义，下面将分别来介绍。

（1）第 1 个取值 same_frame 定义如下：

```
same_frame {
```

```
    u1 frame_type = SAME; /* 0-63 */
}
```

它表示当前代码所在位置和上一个比较位置的局部变量表是完全相同的，并且操作数栈为空。它的取值为 0-63，这个取值也是隐含的 offset_delta，表示距离上一个帧块的偏移量。

（2）第 2 个取值 same_locals_1_stack_item_frame 的定义如下：

```
same_locals_1_stack_item_frame {
    u1 frame_type = SAME_LOCALS_1_STACK_ITEM; /* 64-127 */
    verification_type_info stack[1];
}
```

其中，frame_type 的范围为 64-127，如果栈映射帧为该值，则表示当前帧和上一帧有相同的局部变量，并且操作数栈中变量数量为 1。它有一个隐式的 offset_delta，使用 frame_type – 64 可以计算得来。之后的 verification_type_info，就表示该操作数中的变量类型。

（3）第 3 个取值 same_locals_1_stack_item_frame_extended，和 same_locals_1_stack_item_frame 含义相同，但是前者表示的 offset_delta 范围非常有限，如果超出范围，则需要使用 same_locals_1_stack_item_frame_extended，same_locals_1_stack_item_frame_extended 使用显式的 offset_delta。同样，在结构的最后，存放着操作数栈的数据类型，它的结构如下：

```
same_locals_1_stack_item_frame_extended {
    u1 frame_type = SAME_LOCALS_1_STACK_ITEM_EXTENDED; /* 247 */
    u2 offset_delta;
    verification_type_info stack[1];
}
```

（4）第 4 个取值 chop_frame 则表示操作数栈为空，当前局部变量表比前一帧少 k 个局部变量。其中，k 为 251-frame_type。它的结构如下：

```
chop_frame {
    u1 frame_type = CHOP; /* 248-250 */
    u2 offset_delta;
}
```

（5）第 5 个取值 same_frame_extended 和 same_frame 含义一样，表示局部变量信息和上一帧相同，且操作数栈为空。但是 same_frame_extended 显示指定了 offset_delta，可以表示更大的字节偏移量。它的结构如下：

```
same_frame_extended {
    u1 frame_type = SAME_FRAME_EXTENDED; /* 251 */
    u2 offset_delta;
}
```

（6）第 6 个取值 append_frame 表示当前帧比上一帧多了 k 个局部变量，且操作数栈为空。其中 k 为 frame_type – 251。在 append_frame 的最后，还存放着增加的局部变量的类型。它的结构如下：

```
append_frame {
    u1 frame_type = APPEND; /* 252-254 */
    u2 offset_delta;
    verification_type_info locals[frame_type - 251];
}
```

（7）以上类型均只保存了前后两个帧中变化的部分，因此可以减少数据的大小。但是，如果以上结构都无法表达帧的信息时，则可以使用第 7 种结构 full_frame。它不用来表示连续两个帧之间的差异，而是将局部变量表和操作数栈都做了完整的记录。它的结构如下：

```
full_frame {
    u1 frame_type = FULL_FRAME; /* 255 */
    u2 offset_delta;
    u2 number_of_locals;
    verification_type_info locals[number_of_locals];
    u2 number_of_stack_items;
    verification_type_info stack[number_of_stack_items];
}
```

可以看到，在 full_frame 中，显示指定了 offset_delta，完整记录了局部变量表的数量（number_of_locals）、局部变量表的数据类型（locals）、操作数栈的数量（number_of_stack_items）和操作数栈的类型（stack）。

【示例 9-14】为了帮助读者更好的理解 StackMapTable。这里给出一个简单的示例，查看以下代码：

```
05 public static void append(){
06     int i=0;
07    int j=0;
08    if(i>0){
09        i++;
10    }
11 }
```

由于代码为节选，左侧的行号为该函数在源文件中的实际行号。对于这段代码，可以得到相关的 LineNumberTable、LocalVariableTable 和 StackMapTable，如下所示：

```
LineNumberTable:
 line 6: 0
 line 7: 2
 line 8: 4
 line 9: 8
 line 11: 11

LocalVariableTable:
 Start  Length  Slot  Name   Signature
 2      10      0     i      I
 4      8       1     j      I

StackMapTable: number_of_entries = 1
 frame_type = 253 /* append */
   offset_delta = 11
   locals = [ int, int ]
```

从 StackMapTable 信息中可知，有一条栈映射帧的记录。该记录表明帧类型为 append，偏移量为 11。查 LineNumberTable 得偏移量 11 对应代码第 11 行，即函数结尾，到函数结束时，局部变量表中的数据比前一帧多 253-251=2 个局部变量，且变量类型均为 int 型（就是本例的 i 和 j）。操作数栈中没有数据。这里所说的前一帧，是指函数调用时的局部变量信息。当函数调用时，由于 append()函数没有参数，且为静态函数，故局部变量表为空。

【示例 9-15】栈映射帧的差异可能是增加了局部变量，也有可能减少局部变量。减少局部变量用 chop_frame 表示。下面再看一个减少局部变量的情况：

```
13 public static void chop(){
14     int i = 0;
15     int j = 0;
16     if (i > 0) {
17       long k = 0;
18       if(j==0){
19           k++;
20       }
21       int t=0;
22
23     }
24 }
```

以上代码中的行号，依然是该函数在源文件中的实际行号。编译后，可得如下信息：

```
LineNumberTable:
  line 14: 0
  line 15: 2
  line 16: 4
  line 17: 8
  line 18: 10
  line 19: 14
  line 21: 18
  line 24: 21
LocalVariableTable:
  Start  Length  Slot  Name   Signature
      2      20     0     i   I
      4      18     1     j   I
     10      11     2     k   J
StackMapTable: number_of_entries = 2
     frame_type = 254 /* append */
    offset_delta = 18
    locals = [ int, int, long ]
       frame_type = 250 /* chop */
      offset_delta = 2
```

具体查看 StackMapTable，可以看到，第一条帧记录显示在字节码偏移量 18 的位置比上以帧多了 3 个局部变量，类型分别是 int、int、long，即代码中的 i、j、k。查 LineNumberTable 得该位置为源码第 21 行（不含）。接着，StackMapTable 中的第 2 帧记录显示为 chop，即比上一帧少了 251-250=1 个局部变量，缺少的局部变量即为 long 型的 k。计算对应的代码偏移量可得：18+2+1=21，查找 LineNumberTable，对应代码行数第 24 行，即在第 24 行时，局变量表的数量比第 21 行少了 1 个末尾的 long 型。由源码可知，在第 24 行，局部变量 k 和 t 均已过了作用域，因此不在局部变量表中。

9.2.12　Code 属性总结

由于 Code 属性较为复杂，故将其作用与结构整理成如图 9.14 所示的样子，希望能加强读者记忆。

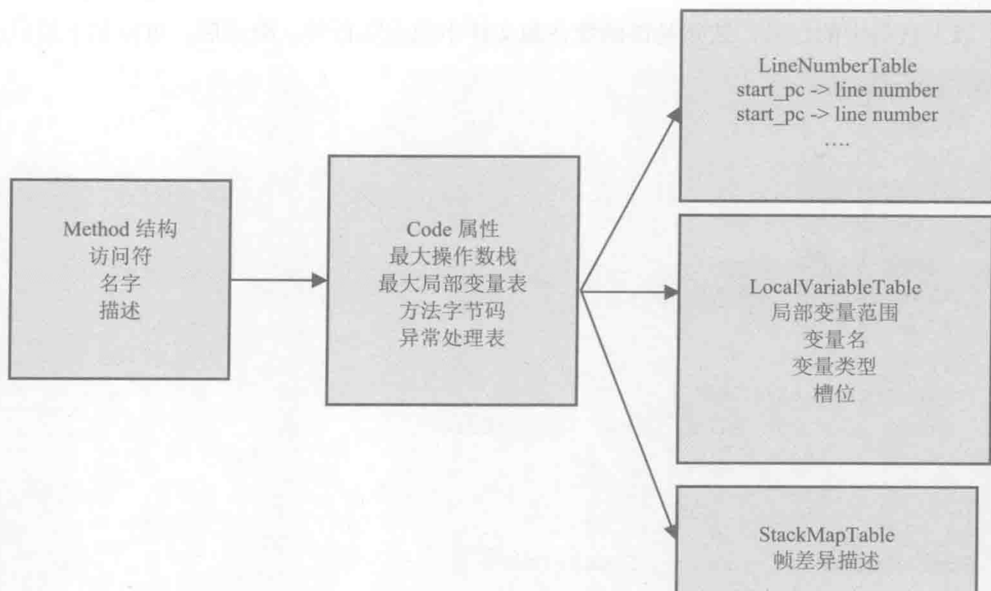


图 9.14 Code 属性常用内容

9.2.13 抛出异常——Exceptions 属性

除了 Code 属性外，每一个方法都可以有一个 Exceptions 属性，用于保存该方法可能抛出的异常信息。该属性的结构如下：

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

Exceptions 与 Code 属性中的异常表不同。Exceptions 属性表示一个方法可能抛出的异常，通常是由方法的 throws 关键字指定的。而 Code 属性中的异常表，则是异常处理机制，由 try-catch 语句生成。

【示例 9-16】比如下面的代码将生成带有 Exceptions 属性的方法。

```
public static void main(String[] args) throws java.io.IOException{

}
```

Exceptions 属性表中，attribute_name_index 指定了属性的名称，它为指向常量池的索引，恒为“Exceptions”，attribute_length 表示属性长度，number_of_exceptions 表示表项数量即可能抛出的异常个数，最后 exception_index_table 项罗列了所有的异常，每一项为指向常量池的索引，对应的常量为 CONSTANT_Class，为一个异常类。

9.2.14 用实例分析 Class 的方法结构

下面将以 SimpleUser 中的 setId()方法为例，详细介绍一下方法结构的具体实现。这里使用 jClasslib 软件来查看 Class 文件的结构，它可以将 Class 文件以可视化的方式进行展示，软件界面如图 9.15 所示。左侧的树形结构显示了 Class 文件的总体概况，包括常量池、接口、字段、方法和属性信息，在左侧结构树中选择，右侧就会显示选中项的详细信息。

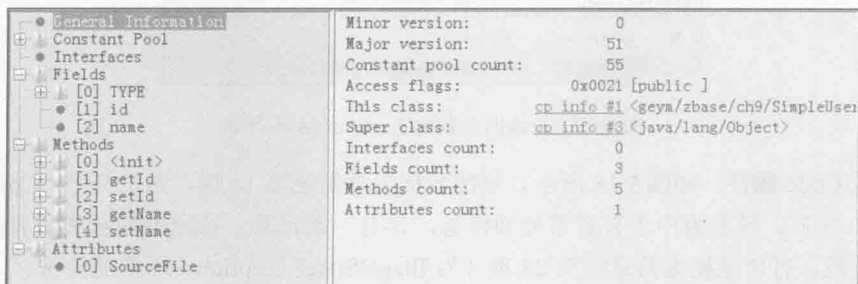


图 9.15　jClasslib 软件界面

为使读者阅读方便，现将 setId()方法重复如下：

```
public void setId(int id) throws IllegalStateException {
    try {
        this.id = id;
    } catch (IllegalStateException e) {
        System.out.println(e.toString());
    }
}
```

该方法将生成如图 9.16 所示的结构。它含有两个属性 Exceptions 和 Code。在 Code 属性中包含了 LineNumberTable、LocalVariableTable 和 StackMapTable 等属性。方法的名称由常量池第 25 项字符串决定，为“setId”。方法的描述为常量池第 26 项字符串为“(I)V”，表示该方法接收一个整数参数，返回值为 void。方法的访问标记为 0x0001 表示这是一个 public 方法。

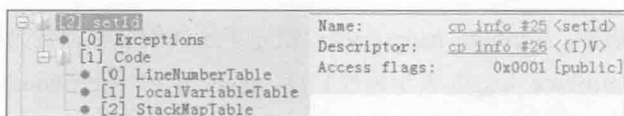


图 9.16 setId()方法

查看方法的 Exceptions 属性可以看到如图 9.17 所示的属性表。属性表的名称为常量池第 27 项，为字符串“Exceptions”。属性长度为 4 字节（不含属性名称和属性长度项）。属性表中只有一条记录，表示 setId()方法可能抛出的异常。异常类型由常量池第 28 项决定，该项为 CONSTANT_Class，表示 IllegalStateException 异常。

Generic info:

Attribute name index: cp info #27

Attribute length: 4

Specific info:

Nr.	exception	verbose
0	cp info #28	java/lang/IllegalStateException

图 9.17 setId()函数的 Exceptions 属性表

再查看 Code 属性，如图 9.18 所示。属性名称为常量池第 14 项，为字符串“Code”。属性长度为 115 字节。异常表中含有异常处理信息，含有一条记录。该记录表示从字节码偏移量 0 到 5 的代码段，有可能抛出常量池第 28 项（为 IllegalStateException）所指的异常。如果在这段代码中遇到这个异常，则转向 handler_pc 进行处理，这里为第 8 个字节码。此外，图中还显示，最大栈深度为 2，最大局变量表为 3，代码长度为 20 字节。

Generic info:

Attribute name index: cp info #14

Attribute length: 115

Specific info:

Bytecode | Exception table | Misc

Nr.	start_pc	end_pc	handler_pc	catch_type	verbose
0	0	5	8	cp info #28	java/lang/IllegalStateException

Specific info:

Bytecode | Exception table | Misc

Maximum stack depth: 2

Maximum local variables: 3

Code length: 20

图 9.18 setId()函数的 Code 属性

查看 LineNumberTable 属性，该属性如图 9.19 所示。属性名称由常量池第 17 项确定，为

字符串“LineNumberTable”，属性长度为 18 字节。一共有 4 条记录，分别表示字节码偏移量与行号的对应关系。可以看到，字节码的起始位置（偏移量为 0）对应的行号为源码第 15 行。

Generic info:

Attribute name index: cp info #17

Attribute length: 18

Specific info:

Nr.	start_pc	line_number
0	0	15
1	5	16
2	9	17
3	19	19

图 9.19　setId()函数的 LineNumberTable 属性

查看 LocalVariableTable 属性，如图 9.20 所示。属性名由常量池第 18 项指定，为字符串“LocalVariableTable”，属性长度为 32 字节。表中显示了该函数的局部变量表。可以看到，该函数拥有 3 个局部变量，其中两个局部变量 this 和 id 的作用范围为 0 到 19（0+20-1），this 为类型 SimpleUser，id 为整数，它们分别占用局部变量的第 0 个和第 1 个槽位。第 3 个局部变量 e 的作用范围为 9 到 18（9+10-1），它是一个 IllegalStateException 类型的变量。

Generic info:

Attribute name index: cp info #18

Attribute length: 32

Specific info:

Nr.	start_pc	length	index	name	descriptor
0	0	20	0	cp info #19 this	cp info #20 Lgeym/zbase/ch9/SimpleUser;
1	0	20	1	cp info #9 id	cp info #8 I
2	9	10	2	cp info #46 e	cp info #47 Ljava/lang/IllegalStateExcept...

图 9.20　setId()函数的 LocalVariableTable 属性

最后来看一下 setId()函数的 StackMapTable 属性。

```
StackMapTable: number_of_entries = 2
    frame_type = 72 /* same_locals_1_stack_item */
  stack = [ class java/lang/IllegalStateException ]
    frame_type = 10 /* same */
```

可以看到，一共两条栈映射帧的记录，第一条记录类型为 same_locals_1_stack_item，表示在字节码偏移量 72-64=8 的位置（不含），局变量表和初始帧相同（局部变量表内为 this 和 id），但是操作数栈中有一个 IllegalStateException 对象，该位置为 catch 语句块内部。第 2 帧为 same 类型，表示在字节码偏移量 72-64+10+1=19 的位置，即函数末尾，局部变量表数据与上一帧相

同，且操作数栈为空。

注意： 此处字节码位置 8 处的语句为 astore_2，表示将栈顶元素存储到第 3 个局部变量中（即变量 e），而此时栈顶元素类型为 IllegalStateException，为异常发生时，系统压入操作数栈的数据。

最后，给出 setId()方法的部分二进制分析，如图 9.21 所示。由于该方法内容较多，因此，很难将所有内容一一在图中呈现，这里只显示其主体部分，余下部分，读者可自行分析。图中从 0x0001 开始，表示该方法为 public 方法；0x0019 为十进制 25，表示方法名由第 25 项常量池决定，为值“setId”；0x001A 表示方法签名，为值“(I)V”。0x0002 表示属性个数，即该方法拥有两个属性 Code 和 Exceptions。0x001B 表示 Exceptions 属性，后续的 0x001C 表示 IllegalStateException 异常。0x000E 表示 Code 属性，0x0002 表示最大操作数栈深度为 2，0x0003 表示最大局部变量表为 3。最后，用细实线包围部分，为该方法的字节码。其余部分限于篇幅不再分析，有兴趣的读者可以自行研究。

```
00000300  00 00 0C 00 01 00 00 00  05 00 13 00 14 00 00 00
00000310  01 00 19 00 1A 00 02 00  1B 00 00 00 04 00 01 00
00000320  1C 00 0E 00 00 00 73 00  02 00 03 00 00 00 14 2A
00000330  1B B5 00 17 A7 00 0E 4D  B2 00 1E 2C B6 00 24 B6
00000340  00 28 B1 00 01 00 00 00  05 00 08 00 1C 00 03 00
00000350  11 00 00 00 12 00 04 00  00 00 0F 00 05 00 10 00
00000360  09 00 11 00 13 00 13 00  12 00 00 00 20 00 03 00
```

图 9.21　setId()方法部分字节码分析

9.2.15　我来自哪里——SourceFile 属性

SourceFile 属性是属于 Class 文件的属性。它用于描述当前这个 Class 文件是由哪个源代码文件编译得来的，格式如下：

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

其中，attribute_name_index 表示属性名称，为指向常量池的一个索引，这里恒为“SourceFile”。attribute_length 为属性长度，对于 SourceFile 属性来说，恒为 2。最后的 sourcefile_

index 表示源代码文件名，它是指向常量池的索引，为 CONSTANT_Utf8 类型。

【示例 9-17】如图 9.22 所示，SourceFile 属性属于 Class 文件的属性，它指向常量池第 54 项，该值为“SimpleUser.java”，表示该 Class 文件的源文件为 SimpleUser.java。

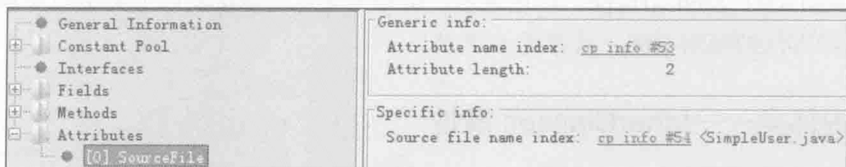


图 9.22　SourceFile 属性

9.2.16　强大的动态调用——BootstrapMethods 属性

为了支持 JDK1.7 中的 invokeDynamic 指令，Java 虚拟机增加了 BootstrapMethods 属性，它用于描述和保存引导方法。引导方法可以理解成是一个查找方法的方法，invokeDynamic 需要能够在运行时根据实际情况返回合适的方法调用，而使用何种策略去查找所需要的方法，是由引导方法决定的。这里的 BootstrapMethods 属性就是用于找到调用的目标方法。该属性是 Class 文件的属性，属性结构如下：

```
BootstrapMethods_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_bootstrap_methods;
    {   u2 bootstrap_method_ref;
        u2 num_bootstrap_arguments;
        u2 bootstrap_arguments[num_bootstrap_arguments];
    } bootstrap_methods[num_bootstrap_methods];
}
```

第 1 个字段 attribute_name_index 表示属性名称，为指向常量池的索引。在这里为“BootstrapMethods”。第 2 个字段 attribute_length 为 4 字节码数字，表示属性的总长度（不含这前 6 个字节）。第 3 个字段 num_bootstrap_methods 表示这个类中包含的引导方法的个数。之后就是 num_bootstrap_methods 个 bootstrap_methods 引导方法。

每一个引导方法又由 3 个字段构成，含义如下。

- bootstrap_method_ref：必须是指向常量池的常数，并且入口为 CONSTANT_MethodHandle，用于指名函数。
- num_bootstrap_arguments：指明引导方法的参数个数。

- bootstrap_arguments：引导方法的参数类型。这是一个指向常量池的索引，且常量池入口只能是：CONSTANT_String、CONSTANT_Class、CONSTANT_Integer、CONSTANT_Long、CONSTANT_Float、CONSTANT_Double、CONSTANT_MethodHandle、或者CONSTANT_MethodType。这也表示，引导方法也只能接受以上类型的参数。

有关引导方法的使用案例，请参考 11.5 节。

9.2.17 内部类——InnerClasses 属性

InnerClass 属性是 Class 文件的属性，它用来描述外部类和内部类之间的联系，其结构如下：

```
InnerClasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    {   u2 inner_class_info_index;
        u2 outer_class_info_index;
        u2 inner_name_index;
        u2 inner_class_access_flags;
    } classes[number_of_classes];
}
```

其中，attribute_name_index 表示属性名称，为指向常量池的索引，这里恒为“InnerClasses”。attribute_length 为属性长度，number_of_classes 表示内部类的个数。classes[number_of_classes] 为描述内部类的表格，每一条内部类记录包含 4 个字段，其中，inner_class_info_index 为指向常量池的指针，它指向一个 CONSTANT_Class，表示内部类的类型。outer_class_info_index 表示外部类类型，也是常量池的索引。inner_name_index 表示内部类的名称，指向常量池中的 CONSTANT_Utf8 项。最后的 inner_class_access_flags 为内部类的访问标识符，用于指示 static、public 等属性，如表 9.10 所示。

表 9.10 常用属性Attribute

访问标记	值	含 义
ACC_PUBLIC	0x0001	public公告类
ACC_PRIVATE	0x0002	私有类
ACC_PROTECTED	0x0004	受保护的类
ACC_STATIC	0x0008	静态内部类
ACC_FINAL	0x0010	final类
ACC_INTERFACE	0x0200	接口
ACC_ABSTRACT	0x0400	抽象类

续表

访问标记	值	含　义
ACC_SYNTHETIC	0x1000	编译器产生的，非代码产生的类
ACC_ANNOTATION	0x2000	注释
ACC_ENUM	0x4000	枚举

【示例 9-18】以下代码使用了内部类。

```
public class SimpleInnerClass {
    public static class In{

    }
}
```

在生成的 Class 文件中，就会有内部类的属性存在，如图 9.23 所示。

General Information
Constant Pool
Interfaces
Fields
Methods
Attributes
[0] SourceFile
[1] InnerClasses

Generic info:
Attribute name index: cp info #16
Attribute length: 10

Specific info:

Nr.	inner_class	outer_class	inner_name	access flags
0	cp info #17 geym/zbase/ch9/SimpleInnerClass$In	cp info #1 geym/zbase/ch9/SimpleInnerClass	cp info #19 In	0x0009 [public static]

图 9.23　内部类属性

该内部类属性中，包含一条内部类数据，说明该类含有一个内部类。内部类 inner_class 指向常量池第 17 项，outer_class 指向常量池第 1 项（即当前类本身），inner_name 为内部类名称 In，访问标示符为 0x0009，即 ACC_STATIC|ACC_PUBLIC，表示一个 public static 的类型。

9.2.18　将要废弃的通知——Deprecated 属性

Deprecated 属性可以用在类、方法、字段等结构中，用于表示该类、方法或者字段将在未来版本中被废弃。Deprecated 属性的结构如下：

```
Deprecated_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

其中，attribute_name_index 表示属性名，为指向常量池的索引，在这里恒为“Deprecated”，attribute_length 为 0。当一个类、方法或者字段被标记为 Depreacted 时，就会产生这个属性。在生成的 SimpleDe 类中，就含有 Deprecated 属性，如下代码所示：

```
@Deprecated
public class SimpleDe {

}
```

9.2.19 Class 文件总结

本节主要介绍了 Class 文件的基本格式。实际上，Class 文件中的内容远不止本节介绍的内容，但限于篇幅有限，不便一一展开。随着 Java 平台的不断发展，在将来，Class 文件的内容也一定会做进一步的扩充，但笔者相信，其基本的格式和结构不会做重大调整。

从 Java 虚拟机的角度看，通过 Class 文件，可以让更多的计算机语言支持 Java 虚拟机平台。因此，Class 文件结构不仅仅是 Java 虚拟机的执行入口，更是 Java 生态圈的基础和核心。

9.3 操作字节码：走进 ASM

ASM 是一款 Java 字节码的操作库，它在 Java 领域是赫赫有名的函数库。不少著名的软件都依赖该库进行字节码操作。比如，AspectJ、Clojure、Eclipse、spring 以及 CGLIB 都是 ASM 的使用者。与 CGLIB 和 Javasssit 等高层字节码库相比，ASM 的性能远远超过它们，由于它直接工作于底层，因此使用也更为灵活，功能也更为强大。但是由于其使用复杂，要求开发人员熟悉和掌握 Class 文件的基本格式和 Java 的字节码，对开发人员要求相对较高，因此很少被直接使用。

但在读者了解了 Class 文件格式和 Java 字节码后，也就有能力直接使用该类库了。因此，本书将介绍和引入 ASM 的使用。

9.3.1 ASM 体系结构

本书以 ASM 5.0 作为基础进行讲解，在 ASM 5.0 中，主要的核心组件如图 9.24 所示。

其中 Opcodes 接口定义了一些常量，尤其是版本号、访问标示符、字节码等信息。ClassReader 用于读取 Class 文件，它的作用是进行 Class 文件的解析，并可以接受一个 ClassVisitor，ClassReader 会将解析过程中产生的类的部分信息，比如访问标示符、字段、方法逐个送入 ClassVisitor，ClassVisitor 在接收到对应的信息后，可以进行各自的处理。

ClassVisitor 有一个重要的子类为 ClassWriter，它负责进行 Class 文件的输出和生成。ClassVisitor 在进行字段和方法处理的时候，会委托给 FieldVisitor 和 MethodVisitor 进行处理。

因此，在类的处理过程中，会创建对应的 FieldVisitor 和 MethodVisitor 对象。FieldVisitor 和 MethodVisitor 类也各自有 1 个重要的子类，FieldWriter 和 MethodWriter。当 ClassWriter 进行字段和方法的处理时，也是依赖这两个类进行的。

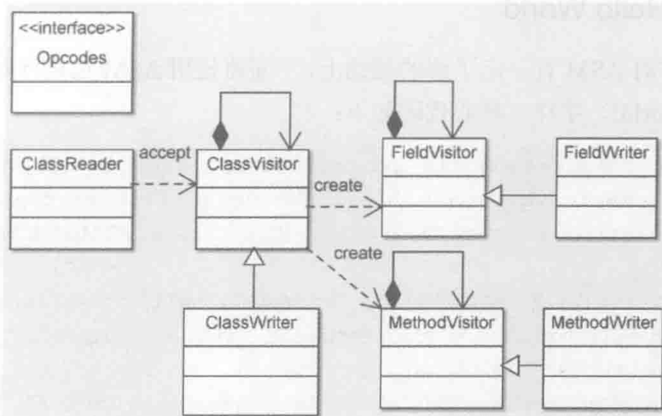


图 9.24　ASM 核心组件

此外，ClassVisitor、FieldVisitor 和 MethodVisitor 都可以使用委托的方式，将实际的处理工作交给内部的委托类进行。它们的内部有一系列的 visitXXX 方法，如图 9.25 所示。



图 9.25　ASM 的 visit 方法族

这些方法在 ClassWriter 和 MethodWriter 内部实现时，在绝大部分情况下，都会去生成该方法对应的内容。比如当 MethodWriter 的 visitInsn(int opcode)方法被调用时，MethodWriter 就会生成一条由参数 opcode 指定的字节码。而 visitInsn(int opcode)方法作为 MethodVisitor 的方法，将会在 ClassReader 访问 Class 时被回调。即，当使用 ClassReader 读取一个类，ClassWriter 作

为访问者时，当 ClassReader 读取到一条不带参数的字节码信息时，就会通知 ClassWriter.visitInsn(int opcode)方法，让 ClassWriter 生成这条字节码信息。

9.3.2 ASM 之 Hello World

【示例 9-19】在对 ASM 有一定了解的基础上，下面将使用 ASM 在运行时动态生成一个类，并打印出“Hello World!”字样。核心代码如下：

```
public class AsmHelloWorld extends ClassLoader implements Opcodes {
    public static void main(final String args[]) throws Exception {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS | ClassWriter.COMPUTE_FRAMES);
        cw.visit(V1_7, ACC_PUBLIC, "Example", null, "java/lang/Object", null);
        MethodVisitor mw = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
        mw.visitVarInsn(ALOAD, 0);
        mw.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "<init>", "()V");
        mw.visitInsn(RETURN);
        mw.visitMaxs(0, 0);
        mw.visitEnd();
        mw = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main", "([Ljava/lang/String;)V", null, null);
        mw.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
        mw.visitLdcInsn("Hello World!");
        mw.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");
        mw.visitInsn(RETURN);
        mw.visitMaxs(0, 0);
        mw.visitEnd();
        byte[] code = cw.toByteArray();

        AsmHelloWorld loader = new AsmHelloWorld();
        Class exampleClass = loader.defineClass("Example", code, 0, code.length);
        exampleClass.getMethods()[0].invoke(null, new Object[] { null });

    }
}
```

首先，为了能够在类生成后，可以较为方便地立即被系统加载，AsmHelloWorld 继承了 ClassLoader，同时，为了能够方便地访问 ASM 的全局常量，也实现了 Opcodes 接口。

在代码第 3 行，创建了 ClassWriter 对象，并指定了 COMPUTE_MAXS 和 COMPUTE_FRAMES 参数。COMPUTE_MAXS 表示希望 ASM 自动计算最大局部变量表和最深操作数栈。这个参数如果不指定，则需要手工计算这两个数值，由于这两个数据需要通过分析字节码流得到，计算过程较为复杂，故设置为自动计算。COMPUTE_FRAMES 表示需要 ASM 自动计算栈映射帧（此标记隐含 COMPUTE_MAXS）。

第 4 行通过 ClassWriter，设置类的基本信息，如访问标记为 public，类名为 Example，父类为 Object。

第 5~10 行生成了 Example 类的构造函数。

第 11~17 行生成 public static void main()方法，并生成了 main()方法的字节码，要求 main()方法运行时调用 System.out.println()方法，输出“Hello World!”。

第 18 行将之前的设置生成 Class 文件流的二进制表示。

第 20~22 行将生成的最终 Class 文件流载入系统，并通过反射调用 main()方法，输出“Hello World!”。

9.4 小结

本章主要介绍了 Class 文件的基本结构，它是 Java 虚拟机的基础。对上层来说，Class 文件屏蔽了开发语言的多样性，作为 Java 虚拟机的唯一接口，结构良好的 Class 文件使得 Java 平台完全有能力支持多种编程语言。对下层来说，依赖于 Java 虚拟机，Class 文件对于不同的计算机平台都是完全统一的，这种多样性的分离使得 Class 文件成为 Java 平台的核心。本章最后还简要介绍了 ASM 开发库，为读者将来通过 ASM 进行复杂的字节码注入打下一个良好的基础。

10

第 10 章 Class 装载系统

上一章我们了解到 Class 文件是虚拟机的入口，那虚拟机是怎样加载这些 Class 文件的呢？Class 文件被载入虚拟机后，又会做哪些额外的处理？类加载的具体步骤是怎么样的？这些都是本章要讨论的重点。

本章涉及的主要知识点有：

- Class 文件的加载过程。
- ClassLoader 的工作模式。
- 有关类的热加载。

10.1 来去都有序：看懂 Class 文件的装载流程

Class 类型通常以文件的形式存在（当然，任何二进制流都可以是 Class 类型），只有被 Java 虚拟机装载的 Class 类型才能在程序中使用。系统装载 Class 类型可以分为加载、连接和初始化

3 个步骤。其中，连接又可分为验证、准备和解析 3 步，如图 10.1 所示。

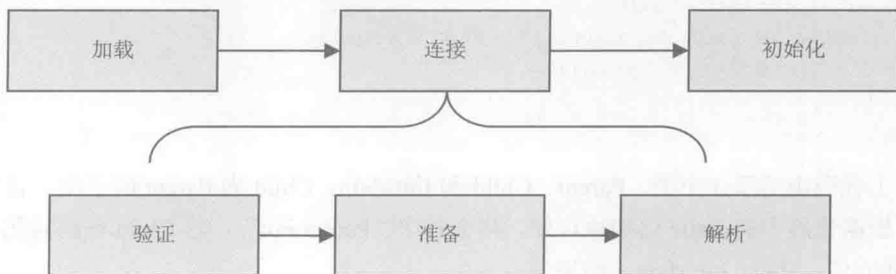


图 10.1　Class 文件装载过程

10.1.1　类装载的条件

Class 只有在必须要使用的时候才会被装载，Java 虚拟机不会无条件地装载 Class 类型。Java 虚拟机规定，一个类或接口在初次使用前，必须要进行初始化。这里指的“使用”，是指主动使用，主动使用只有下列几种情况：

- 当创建一个类的实例时，比如使用 new 关键字，或者通过反射、克隆、反序列化。
- 当调用类的静态方法时，即当使用了字节码 invokestatic 指令。
- 当使用类或接口的静态字段时（final 常量除外），比如，使用 getstatic 或者 putstatic 指令。
- 当使用 java.lang.reflect 包中的方法反射类的方法时。
- 当初始化子类时，要求先初始化父类。
- 作为启动虚拟机，含有 main()方法的那个类。

除了以上的情况属于主动使用，其他的情况均属于被动使用。被动使用不会引起类的初始化。

【示例 10-1】下面先来看一个主动引用的例子。

```
public class Parent {
    static{
        System.out.println("Parent init");
    }
}

public class Child extends Parent{
    static{
        System.out.println("Child init");
    }
}
```

```
public class InitMain {
    public static void main(String[] args) {
        Child c=new Child();
    }
}
```

以上代码申明了 3 个类，Parent、Child 和 InitMain，Child 为 Parent 的子类。若 Parent 被初始化，根据代码中的 static 语句块可知，将会打印“Parent init”，若 Child 被初始化，则会打印“Child init”。执行 InitMain，结果为：

```
Parent init
Child init
```

由此可知，系统首先装载 Parent 类，接着再装载 Child 类。符合主动装载中的两个条件，使用 new 关键字创建类的实例会装载相关类，以及在初始化子类时，必须先初始化父类。

【示例 10-2】主动引用比较容易理解，下面再来看几个被动引用的例子。被动引用不会导致类的装载。

```
public class Parent {
    static {
        System.out.println("Parent init");
    }
    public static int v = 100;
}

public class Child extends Parent{
    static{
        System.out.println("Child init");
    }
}

public class UseParent {
    public static void main(String[] args) {
        System.out.println(Child.v);
    }
}
```

查看以上代码，Parent 中有静态变量 v，并且在 UseParent 中，使用其子类 Child 去调用父类中的变量。运行以上代码，输出结果如下：

```
Parent init
100
```

可以看到，虽然在 UseParent 中，直接访问了子类对象，但是 Child 子类并未被初始化，只有 Parent 父类被初始化。可见，在引用一个字段时，只有直接定义该字段的类，才会被初始化。

注意： 虽然 Child 类没有被初始化，但是，此时 Child 类已经被系统加载，只是没有进入到初始化阶段。

如果使用-XX:+TraceClassLoading 参数运行这段代码，就会得到以下日志（限于篇幅，只列出部分输出）：

```
[Loaded geym.zbase.ch10.noinit.Parent from file:/C:/Users/Administrator/workspace/zbaseJvm/bin/]
[Loaded geym.zbase.ch10.noinit.Child from file:/C:/Users/Administrator/workspace/zbaseJvm/bin/]
Parent init
100
[Loaded java.lang.Shutdown from shared objects file]
```

从这段日志中可以看到，Child 子类确实已经被加载入系统，但是 Child 的初始化却未进行。

【示例 10-3】另外一个有趣的例子是使用引用常量。在前文介绍的几种主动使用的情况中，特别注明："当使用类或接口的静态字段时（final 常量除外）"，也就是说引用 final 常量并不会引起类的初始化。查看以下代码：

```
public class FinalFieldClass {
    public static final String constString="CONST";
    static{
        System.out.println("FinalFieldClass init");
    }
}

public class UseFinalField {
    public static void main(String[] args) {
        System.out.println(FinalFieldClass.constString);
    }
}
```

运行以上代码输出结果为：

```
CONST
```

FinalFieldClass 类并没有因为其常量字段 constString 被引用而初始化。这是因为在 Class 文件生成时，final 常量由于其不变性，做了适当的优化。分析 UseFinalField 类生成的 Class 文件，

可以看到 main()函数的字节码为：

```
   0:  getstatic    #16; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:  ldc    #22; //String CONST
   5:  invokevirtual   #24; //Method java/io/PrintStream.println:(Ljava/lang/
String;)V
   8:  return
```

在字节码偏移 3 的位置，通过 ldc 将常量池第 22 项入栈，在此 Class 文件中常量池第 22 项为：

```
   #22 = String              #23           //  CONST
   #23 = Utf8               CONST
```

可以看到，在编译后的 UseFinalField.class 中，并没有引用 FinalFieldClass 类，而是将其 final 常量直接存放到常量池中，因此，FinalFieldClass 类自然不会被加载。由这段分析可知，在这种情况下，FinalFieldClass 类根本不会被加载到系统中，通过捕获类加载日志即可证明这一点（部分日志）：

```
   [Loaded java.security.BasicPermissionCollection from shared objects file]
   [Loaded geym.zbase.ch10.noinit.UseFinalField from file:/C:/Users/Administrator/
workspace/zbaseJvm/bin/]
   [Loaded java.lang.Void from shared objects file]
   CONST
   [Loaded java.lang.Shutdown from shared objects file]
```

在所有的类加载日志中，没有 FinalFieldClass 类出现，可见，javac 在编译时，将常量直接植入目标类，不再使用被引用类。

注意： 并不是在代码中出现的类，就一定会被加载或者初始化。如果不符合主动使用的条件，类就不会初始化。

10.1.2 加载类

加载类处于类装载的第一个阶段。在加载类时，Java 虚拟机必须完成以下工作：

- 通过类的全名，获取类的二进制数据流。
- 解析类的二进制数据流为方法区内的数据结构。
- 创建 java.lang.Class 类的实例，表示该类型。

对于类的二进制数据流，虚拟机可以通过多种途径产生或获得。最一般地，虚拟机可能通过文件系统读入一个 class 后缀的文件，或者也可能读入 JAR、ZIP 等归档数据包，提取类文件。除了这些形式外，任何形式都是可以的。比如，事先将类的二进制数据存放在数据库中，或者通过

类似于 HTTP 之类的协议通过网络进行加载，甚至是在运行时生成一段 Class 的二进制信息。

在获取到类的二进制信息后，Java 虚拟机就会处理这些数据，并最终转为一个 java.lang.Class 的实例，java.lang.Class 实例是访问类型元数据的接口，也是实现反射的关键数据。通过 Class 类提供的接口，可以访问一个类型的方法、字段等信息。

【示例 10-4】以下代码通过 Class 类，获得了 java.lang.String 类的所有方法信息，并打印方法访问标示符以及方法签名。

```
01 public static void main(String[] args) throws Exception {
02     Class clzStr=Class.forName("java.lang.String");
03     Method[] ms=clzStr.getDeclaredMethods();
04     for(Method m:ms){
05         String mod=Modifier.toString(m.getModifiers());
06         System.out.print(mod+" "+ m.getName()+" (");
07         Class<?>[] ps=m.getParameterTypes();
08         if(ps.length==0)System.out.print(')');
09         for(int i=0;i<ps.length;i++){
10             char end=i==ps.length-1?')':',';
11             System.out.print(ps[i].getSimpleName()+end);
12         }
13         System.out.println();
14     }
15 }
```

代码第 2 行，通过 Class.forName()方法得到代表 String 类的 Class 实例。第 3 行通过 Class.getDeclaredMethods()方法取得 String 类的所有方法列表。

第 5 行取得方法的访问标示符，并通过 Modifier.toString()方法将访问标示符转为可读字符串。第 7 行取得方法的所有参数。第 11 行输出方法的参数。

这段代码运行后，部分输出结果如下：

```
省略部分输出
static lastIndexOf (char[],int,int,char[],int,int,int)
public lastIndexOf (String)
private lastIndexOfSupplementary (int,int)
public length ()
public matches (String)
```

在 Java 虚拟机中，完成加载类，尤其是获取类的二进制信息的组件就是 ClassLoader 类加载器，它也是本章后续要讨论的重要主题。

10.1.3 验证类

当类加载到系统后，就开始连接操作，验证是连接操作的第一步。它的目的是保证加载的字节码是合法、合理并符合规范的。验证的步骤比较复杂，实际要验证的项目也很繁多，大体上 Java 虚拟机需要做以下检查，如图 10.2 所示。

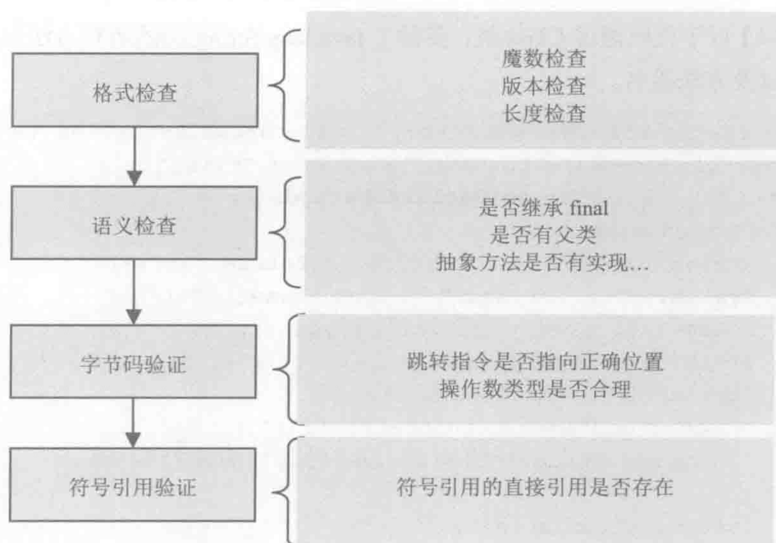


图 10.2 Java 虚拟机验证过程

（1）必须判断类的二进制数据是否是符合格式要求和规范的。比如，是否以魔数 0xCAFEBABE 开头，主版本和小版本号是否在当前 Java 虚拟机的支持范围内，数据中每一个项是否都拥有正确的长度等等。

（2）Java 虚拟机会进行字节码的语义检查，比如是否所有的类都有父类的存在（在 Java 里，除了 Object 外，其他类都应该有父类），是否一些被定义为 final 的方法或者类被重载或继承了，非抽象类是否实现了所有抽象方法或者接口方法，是否存在不兼容的方法（比如方法的签名除了返回值不同，其他都一样，这种方法会让虚拟机无从下手调度），但凡在语义上不符合规范的，虚拟机也不会给予验证通过。

（3）Java 虚拟机还会进行字节码验证，字节码验证也是验证过程中最为复杂的一个过程。它试图通过对字节码流的分析，判断字节码是否可以被正确地执行。比如，在字节码的执行过程中，是否会跳转到一条不存在的指令，函数的调用是否传递了正确类型的参数，变量的赋值是不是给了正确的数据类型等。在本书第 9 章中介绍的栈映射帧（StackMapTable）就是在这个

阶段，用于检测在特定的字节码处，其局部变量表和操作数栈是否有着正确的数据类型。但遗憾的是，100%准确地判断一段字节码是否可以被安全执行是无法实现的，因此，该过程只是尽可能地检查出可以预知的明显的问题。如果在这个阶段无法通过检查，虚拟机也不会正确装载这个类。但是，如果通过了这个阶段的检查，也不能说明这个类是完全没有问题的。

在前面 3 次检查中，已经排除了文件格式错误、语义错误以及字节码的不正确性。但是依然不能确保类是没有问题的。

（4）校验器还将进行符号引用的验证。根据 9.2 节的介绍，Class 文件在其常量池会通过字符串记录自己将要使用的其他类或者方法。因此，在验证阶段，虚拟机就会检查这些类或者方法确实是存在的，并且当前类有权限访问这些数据，如果一个需要使用类无法在系统中找到，则会抛出 NoClassDefFoundError，如果一个方法无法被找到，则会抛出 NoSuchMethodError。

10.1.4　准备

当一个类验证通过时，虚拟机就会进入准备阶段。在这个阶段，虚拟机就会为这个类分配相应的内存空间，并设置初始值。Java 虚拟机为各类型变量默认的初始值如表 10.1 所示。

表 10.1　数据类型和默认初始值对应

类　型	默认初始值
int	0
long	0L
short	(short)0
char	\u0000
boolean	false
reference	null
float	0f
double	0f

注意： Java 并不支持 boolean 类型，对于 boolean 类型，内部实现是 int，由于 int 的默认值是 0，故对应的，boolean 的默认值就是 false。

如果类存在常量字段，那么常量字段也会在准备阶段被附上正确的值，这个赋值属于 Java 虚拟机的行为，属于变量的初始化。事实上，在准备阶段，不会有任何 Java 代码被执行。

【示例 10-5】比如，若在类中定义了以下常量：

```
public static final String constString="CONST";
```

那么，生成的 Class 文件，就会如图 10.3 所示，可以看到该字段含有 ConsantValue 属性，直接存放于常量池中。该常量 constString 在准备阶段被附上字符串“CONST”（并非由 Java 字节码引起的）。

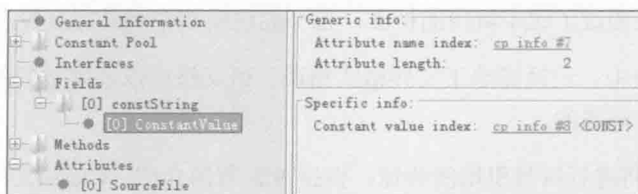


图 10.3　final 常量的定义

但如果没有 final 的修饰，仅仅作为普通的静态变量：

```
public static String constString="CONST";
```

此时，constString 的赋值在函数<clinit>中发生，属于 Java 字节码的行为。如图 10.4 所示。字段 constString 上未携带任何数据信息，在<clinit>方法中，将字符串常量 CONST 通过 ldc 指令压栈，并通过 putstatic 语句进行赋值。

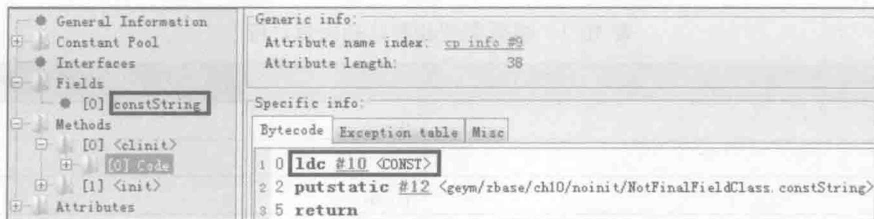


图 10.4　非 final 变量的赋值

注意： ldc 字节码会加载一个常量到操作数栈中，putstatic 字节码设置给定的静态字段的值。

10.1.5　解析类

在准备阶段完成后，就进入了解析阶段。解析阶段的工作就是将类、接口、字段和方法的符号引用转为直接引用。

符号引用就是一些字面量的引用，和虚拟机的内部数据结构和和内存布局无关。比较容易理解的就是在 Class 类文件中，通过常量池进行了大量的符号引用。下面通过一个简单的函数调用来理解一下符号引用是如何工作的。

【示例 10-6】下面的字节码调用了 System.out.println()。

```
invokevirtual #24 <java/io/PrintStream.println>
```

其中，这里使用了常量池第 24 项，查看并分析该常量池，可以看到如图 10.5 所示的结构。

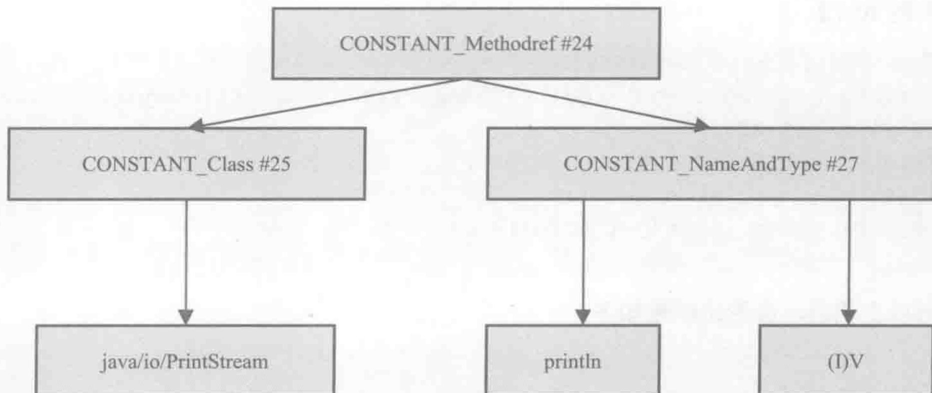


图 10.5　符号引用例子

常量池第 24 项被 invokevirtual 使用，顺着 CONSTANT_Methodref #24 的引用关系查找，最终发现，所有对于 Class 以及 NameAndType 类型的引用都是基于字符串的。因此，可以认为 invokevirtual 的函数调用通过字面量的引用描述已经表达清楚。这就是符号引用。

但是在程序实际运行时，只有符号引用是不够的，当 println()方法被调用时，系统需要明确知道该方法的位置。以方法为例，Java 虚拟机为每个类都准备了一张方法表，将其所有的方法都列在表中，当需要调用一个类的方法的时候，只要知道这个方法在方法表中的偏移量就可以直接调用该方法。通过解析操作，符号引用就可以转变为目标方法在类中方法表中的位置，从而使得方法被成功调用。

综上所述，所谓解析就是将符号引用转为直接引用，也就是得到类或者字段、方法在内存中的指针或者偏移量，因此，可以说，如果直接引用存在，那么可以肯定系统中存在该类、方法或者字段。但只存在符号引用，不能确定系统中一定存在该对象。

最后，再来看一下 CONSTANT_String 的解析。由于字符串在程序开发中有着重要的作用，因此，读者有必要了解一下 String 在 Java 虚拟机中的处理。当在 Java 代码中直接使用字符串常量时，就会在类中出现 CONSTANT_String，它表示字符串常量，并且会引用一个 CONSTANT_UTF8 的常量项。在 Java 虚拟机内部运行时的常量池中，会维护一张字符串拘留表，它会保存所有出现过的字符串常量，并且没有重复项。只要以 CONSTANT_String 形式出

现的字符串也都会在这张表中。使用 String.intern()方法可以得到一个字符串在拘留表（intern）中的引用，因为该表中没有重复项，所以任何字面相同的字符串的 String.intern()方法返回总是相等的。

【示例 10-7】

```
public static void main(String[] args) {
    String a=Integer.toString(1)+Integer.toString(2)+Integer.toString(3);
    String b="123";
    System.out.println(a.equals(b));
    System.out.println(a==b);
    System.out.println(a.intern()==b);
}
```

运行以上代码，会输出结果如下：

```
true
false
true
```

因为 a 和 b 两个字符串都是字面量“123”，所以 equals()方法返回 true，而 a 和 b 并不是指向同一个对象，因此 a==b 返回 false。但是由于 a 在拘留表（intern）中的引用，就是 b（这里 b 就是常量本身），因此 a.intern()==b 返回 true。

10.1.6 初始化

类的初始化是类装载的最后一个阶段。如果前面的步骤都没有问题，那么表示类可以顺利装载到系统中。此时，类才会开始执行 Java 字节码。初始化阶段的重要工作是执行类的初始化方法<clinit>。方法<clinit>是由编译器自动生成的，它是由类静态成员的赋值语句以及 static 语句块合并产生的。

【示例 10-8】请查看以下代码：

```
public class SimpleStatic {
    public static int id=1;
    public static int number;
    static{
        number=4;
    }
}
```

Java 编译器会为这段代码生成如下的<clinit>：

```
0:   iconst_1
1:   putstatic       #11; //Field id:I
4:   iconst_4
5:   putstatic       #13; //Field number:I
8:   return
```

可以看到，生成的<clinit>函数中，整合了 SimpleStatic 类中的 static 赋值语句以及 static 语句块，先后对 id 和 number 两个成员变量进行赋值。

由于在加载一个类之前，虚拟机总是会试图加载该类的父类，因此父类的<clinit>总是在子类<clinit>之前被调用。也就是说，子类的 static 块优先级高于父类。

【示例 10-9】再来看以下代码：

```
public class ChildStatic extends SimpleStatic{
    static{
        number=2;
    }
    public static void main(String[] args) {
        System.out.println(number);
    }
}
```

这段代码集成了 SimpleStatic 类，并且在其 static 语句块中重新为 number 变量赋值为 2。在 main()方法中，输出 number 变量，结果为 2，由此可见，ChildStatic 类的 static 语句块覆盖了 SimpleStatic 中的 static 语句块。

但 Java 编译器并不会为所有的类都产生<clinit>初始化函数。如果一个类既没有赋值语句，也没有 static 语句块，那么生成的<clinit>函数就应该为空，因此，编译器就不会为该类插入<clinit>函数。

【示例 10-10】比如以下类：

```
public class StaticFinalClass {
    public static final int i=1;
    public static final int j=2;
}
```

由于 StaticFinalClass 只有 final 常量，而 final 常量在准备阶段初始化，而并不在初始化阶段处理，因此对于 StaticFinalClass 来说，<clinit>就无事可做，因此，在产生的 class 文件中，没有该函数存在。

最后值得一提的是，对于<clinit>函数的调用，也就是类的初始化，虚拟机会在内部确保其多线程环境中的安全性，也就是说，当多个线程试图初始化同一个类时，只有一个线程可以进入<clinit>函数，而其他线程必须等待，如果之前的线程成功加载了类，则等在队列中的线程就没有机会再执行<clinit>函数了（当需要使用这个类时，虚拟机会直接返回给它已经准备好的信息）。

正是因为函数<clinit>是带锁线程安全的，因此，在多线程环境下进行类初始化的时候，可能会引起死锁，并且这种死锁是很难发现的，因为看起来它们并没有可用的锁信息。

【示例 10-11】下面代码展示了在类初始化的时候，产生了线程死锁问题。

```
public class StaticA {
    static{
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        try {
            Class.forName("geym.zbase.ch10.staticdead.StaticB");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        System.out.println("StaticA init OK");
    }
}

public class StaticB {
    static{
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        try {
            Class.forName("geym.zbase.ch10.staticdead.StaticA");
        } catch (ClassNotFoundException e) {
        }
        System.out.println("StaticB init OK");
    }
}

public class StaticDeadLockMain extends Thread{
    private char flag;
```

```
    public StaticDeadLockMain(char flag){
        this.flag=flag;
        this.setName("Thread"+flag);
    }
    @Override
    public void run(){
        try {
            Class.forName("geym.zbase.ch10.staticdead.Static"+flag);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        System.out.println(getName()+" over");
    }
    public static void main(String[] args) throws InterruptedException {
        StaticDeadLockMain loadA=new StaticDeadLockMain('A');
        loadA.start();
        StaticDeadLockMain loadB=new StaticDeadLockMain('B');
        loadB.start();
    }
}
```

以上代码由 3 个类组成，StaticA、StaticB 和 StaticDeadLockMain。在 StaticDeadLockMain 中创建了两个线程，线程 A 试图去初始化 StaticA，线程 B 尝试去初始化 StaticB，在 StaticA 的初始化过程中，会去尝试初始化 StaticB，同样在 StaticB 的初始化过程中，也去初始化 StaticA。如图 10.6 所示，这种情况导致了系统死锁。

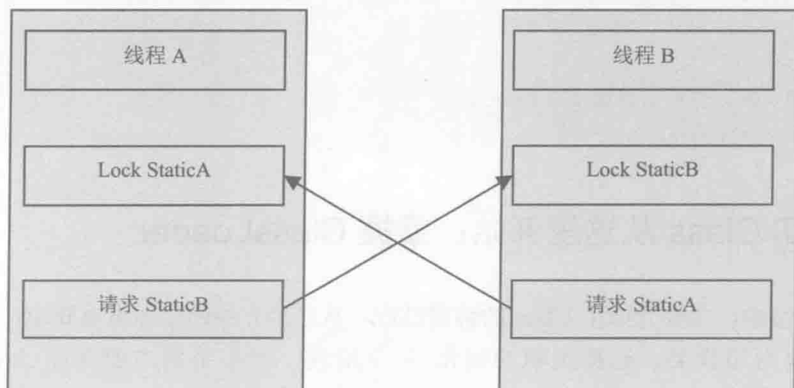


图 10.6　类初始化函数死锁示意图

通过线程堆栈 Dump 可以得到如下信息，可以看到在这种情况下，系统并没有给出足够的

信息来判定死锁，但是死锁的情况确实存在，因此需要格外小心由类的初始化引起的死锁问题。

```
"ThreadB" prio=6 tid=0x000000000aeb0000 nid=0xed2c in Object.wait()
[0x000000000c92e000]
    java.lang.Thread.State: RUNNABLE
     at java.lang.Class.forName0(Native Method)
     at java.lang.Class.forName(Class.java:190)
     at geym.zbase.ch10.staticdead.StaticB.<clinit>(StaticB.java:10)
     at java.lang.Class.forName0(Native Method)
     at java.lang.Class.forName(Class.java:190)
     at geym.zbase.ch10.staticdead.StaticDeadLockMain.run(StaticDeadLockMain.
java:12)

    Locked ownable synchronizers:
     - None

"ThreadA" prio=6 tid=0x000000000aeaf000 nid=0xfff0 in Object.wait()
[0x000000000c6ae000]
    java.lang.Thread.State: RUNNABLE
     at java.lang.Class.forName0(Native Method)
     at java.lang.Class.forName(Class.java:190)
     at geym.zbase.ch10.staticdead.StaticA.<clinit>(StaticA.java:10)
     at java.lang.Class.forName0(Native Method)
     at java.lang.Class.forName(Class.java:190)
     at geym.zbase.ch10.staticdead.StaticDeadLockMain.run(StaticDeadLockMain.
java:12)

    Locked ownable synchronizers:
     - None
```

10.2 一切 Class 从这里开始：掌握 ClassLoader

在前面一节中，主要介绍了 Class 的装载过程，从已经介绍的知识可以知道，Class 的装载大体上可以分为加载类、连接类和初始化 3 个阶段。本小节将主要介绍 Java 语言中的 ClassLoader，类装载器。ClassLoader 在 Java 中有着非常重要的作用，它主要工作在 Class 装载的加载阶段，其主要作用是从系统外部获得 Class 二进制数据流。

10.2.1　认识 ClassLoader，看懂类加载

ClassLoader 是 Java 的核心组件，所有的 Class 都是由 ClassLoader 进行加载的，ClassLoader 负责通过各种方式将 Class 信息的二进制数据流读入系统，然后交给 Java 虚拟机进行连接、初始化等操作。因此，ClassLoader 在整个装载阶段，只能影响到类的加载，而无法通过 ClassLoader 去改变类的连接和初始化行为。

从代码层面看，ClassLoader 是一个抽象类，它提供了一些重要的接口，用于自定义 Class 的加载流程和加载方式。ClassLoader 的主要方法如下：

- public Class<?> loadClass(String name) throws ClassNotFoundException
  给定一个类名，加载一个类，返回代表这个类的 Class 实例，如果找不到类，则返回 ClassNotFoundException 异常。
- protected final Class<?> defineClass(byte[] b, int off, int len)
  根据给定的字节码流 b 定义一个类，off 和 len 参数表示实际 Class 信息在 byte 数组中的位置和长度，其中 byte 数组 b 是 ClassLoader 从外部获取的。这是受保护的方法，只有在自定义 ClassLoader 子类中可以使用。
- protected Class<?> findClass(String name) throws ClassNotFoundException
  查找一个类，这是一个受保护的方法，也是重载 ClassLoader 时，重要的系统扩展点。这个方法会在 loadClass()时被调用，用于自定义查找类的逻辑。如果不需要修改类加载默认机制，只是想改变类加载的形式，就可以重载该方法。
- protected final Class<?> findLoadedClass(String name)
  这也是一个受保护的方法，它会去寻找已经加载的类。这个方法是 final 方法，无法被修改。

在 ClassLoader 的结构中，还有一个重要的字段 parent，它也是一个 ClassLoader 的实例，这个字段所表示的 ClassLoader 也称为这个 ClassLoader 的双亲。在类加载的过程中，ClassLoader 可能会将某些请求交予自己的双亲处理。

10.2.2　ClassLoader 的分类

在标准的 Java 程序中，Java 虚拟机会创建 3 类 ClassLoader 为整个应用程序服务。它们分别是：BootStrap ClassLoader（启动类加载器）、Extension ClassLoader（扩展类加载器）和 App ClassLoader（应用类加载器，也称为系统类加载器）。此外，每一个应用程序还可以拥有自定义的 ClassLoader，扩展 Java 虚拟机获取 Class 数据的能力。

各个 ClassLoader 的层次和功能如图 10.7 所示，从 ClassLoader 的层次自顶往下为启动类加载器、扩展类加载器、应用类加载器和自定义类加载器。其中，应用类加载器的双亲为扩展类加载器，扩展类加载器的双亲为启动类加载器。当系统需要使用一个类时，在判断类是否已经被加载时，会先从当前底层类加载器进行判断。当系统需要加载一个类时，会从顶层类开始加载，依次向下尝试，直到成功。

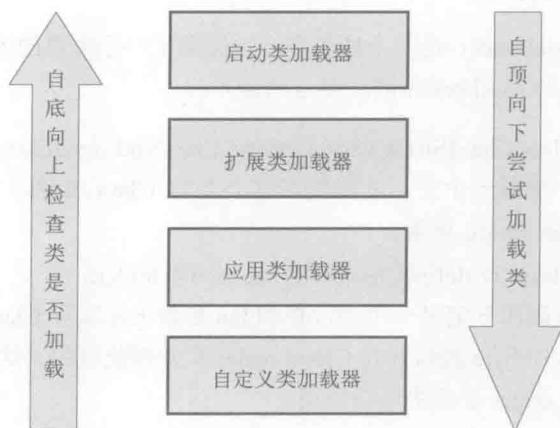


图 10.7　ClassLoader 层次结构

在这些 ClassLoader 中，启动类加载器最为特别，它是完全由 C 代码实现的，并且在 Java 中没有对象与之对应。系统的核心类就是由启动类加载器进行加载的，它也是虚拟机的核心组件。扩展类加载器和应用类加载器都有对应的 Java 对象可供使用。

【示例 10-12】以下代码验证了这个问题，并帮助读者理解各加载器之间的关系。

```
public class PrintClassLoaderTree {
    public static void main(String[] args) {
        ClassLoader cl=PrintClassLoaderTree.class.getClassLoader();
        while(cl!=null){
            System.out.println(cl);
            cl=cl.getParent();
        }
    }
}
```

代码中先取得装载当前类 PrintClassLoaderTree 的 ClassLoader，然后打印当前 ClassLoader 并获得其双亲，直到类加载器树被遍历完成。以上代码输出结果如下：

```
sun.misc.Launcher$AppClassLoader@b23210
```

```
sun.misc.Launcher$ExtClassLoader@f4f44a
```

由此得知，PrintClassLoaderTree 用户类加载于 AppClassLoader（应用类加载器）中，而 AppClassLoader 的双亲为 ExtClassLoader（扩展类加载器）。而从 ExtClassLoader 无法再取得启动类加载器，因为这是一个系统级的纯 C 实现。因此，任何加载在启动类加载器中的类是无法获得其 ClassLoader 实例的，比如：

```
String.class.getClassLoader()
```

由于 String 属于 Java 核心类，因此会被启动类加载器加载，故以上代码返回的是 null。

注意： 无法在 Java 代码中访问启动类加载器，当试图获得一个类的 ClassLoader 时，如果得到的是 null，这并不意味着没有加载器为它服务，而是指加载那个类的为启动类加载器。

在虚拟机设计中，使用这种分散的 ClassLoader 去装载类是有好处的，不同层次的类可以由不同的 ClassLoader 加载，从而进行划分，这有助于系统的模块化设计。一般来说，启动类加载器负责加载系统的核心类，比如 rt.jar 中的 Java 类；扩展类加载器用于加载%JAVA_HOME%/lib/ext/*.jar 中的 Java 类；应用类加载器用于加载用户类，也就是用户程序的类；自定义类加载器用于加载一些特殊途径的类，一般也是用户程序类。

10.2.3　ClassLoader 的双亲委托模式

系统中的 ClassLoader 在协同工作时，默认会使用双亲委托模式。即在类加载的时候，系统会判断当前类是否已经被加载，如果已经被加载，就会直接返回可用的类，否则就会尝试加载，在尝试加载时，会先请求双亲处理，如果双亲请求失败，则会自己加载。

【示例 10-13】以下代码显示了 ClassLoader 加载类的详细过程，它实现在 ClassLoader.loadClass()中。

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
```

```
11                  c = parent.loadClass(name, false);
12              } else {
13                  c = findBootstrapClassOrNull(name);
14              }
15          } catch (ClassNotFoundException e) {
16              // ClassNotFoundException thrown if class not found
17              // from the non-null parent class loader
18          }
19
20          if (c == null) {
21              // If still not found, then invoke findClass in order
22              // to find the class.
23              long t1 = System.nanoTime();
24              c = findClass(name);
25
26              // this is the defining class loader; record the stats
27              sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
28              sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
29              sun.misc.PerfCounter.getFindClasses().increment();
30          }
31      }
32      if (resolve) {
33          resolveClass(c);
34      }
35      return c;
36  }
37 }
```

在代码第 6 行，当前 ClassLoader 试图查找该类是否已经被加载，如果已经被加载则直接返回。如果没有被加载，则会在第 11 行请求其双亲加载（不是自己加载），如果双亲为 null 时，则使用启动类加载器加载。如果双亲加载不成功，则会在 24 行由当前 ClassLoader 尝试加载。

提示： *双亲为 null 有两种情况：第一，其双亲就是启动类加载器；第二，当前加载器就是启动类加载器。*

【示例 10-14】下面来看一个实验，FindClassOrder 创建了一个 HelloLoader 的实例，并调用了 print()。

```
public class FindClassOrder {
    public static void main(String args[]){
```

```
        HelloLoader loader=new HelloLoader();
        loader.print();
    }
}
```

HelloLoader 有两个版本，版本一称为启动版本：

```
public class HelloLoader {
    public void print(){
        System.out.println("I am in Boot ClassLoader");
    }
}
```

版本二称为应用版本：

```
public class HelloLoader {
    public void print(){
        System.out.println("I am in Boot ClassLoader");
    }
}
```

应用版本和 FindClassOrder 都在应用的 ClassPath 下，启动版本的 HelloLoader 放置于笔者计算机的 D:/tmp/clz 目录下。

直接运行 FindClassOrder，得到输出结果如下：

```
I am in apploader
```

这个输出是显而易见的，因为 FindClassOrder 自然会在当前 ClassPath 下查找需要的类，但是如果修改启动 ClassPath，结果就会不一样，增加以下参数来运行这个程序：

```
-Xbootclasspath/a:D:/tmp/clz
```

虚拟机参数-Xbootclasspath 修改了启动 ClassPath，将指定的 D:/tmp/clz 追加到了启动 ClassPath 后，该参数指明的路径下的类，将会被启动类加载器搜索到。带着这个参数运行程序，得到：

```
I am in Boot ClassLoader
```

可以看到程序的输出发生了变化。HelloLoader 的应用版本并没有运行，取而代之的是启动版本（应用版本依然在当前 ClassPath 下）。由此可以推测，当系统需要加载一个类时，会先从顶层的启动类加载器开始加载，逐层往下，直到找到该类。

【示例 10-15】现在，简单地修改一下 FindClassOrder。

```
01 public static void main(String args[]) throws Exception {
02     ClassLoader cl=FindClassOrder2.class.getClassLoader();
03     byte[] bHelloLoader=loadClassBytes("geym.zbase.ch10.findorder.HelloLoader");
04     Method md_defineClass=ClassLoader.class.getDeclaredMethod("defineClass",
byte[].class,int.class,int.class);
05     md_defineClass.setAccessible(true);
06     md_defineClass.invoke(cl, bHelloLoader,0,bHelloLoader.length);
07     md_defineClass.setAccessible(false);
08
09     HelloLoader loader = new HelloLoader();
10     System.out.println(loader.getClass().getClassLoader());
11     loader.print();
12 }
```

代码第 2~7 行，在 HelloLoader 被使用之前，先将应用版本的 HelloLoader，通过 ClassLoader 的 defineClass()方法进行定义。代码第 2 行显示，取得的 ClassLoader 为加载 FindClassOrder2 的应用类加载器，因此第 2~7 行把应用版本的 HelloLoader 手工加载到应用类加载器中。使用虚拟机参数“-Xbootclasspath/a:D:/tmp/clz”运行以上代码，结果为：

```
sun.misc.Launcher$AppClassLoader@7cb9e9a3
I am in apploader
```

可以看到，虽然定义了启动 ClassPath，但是系统并没有加载启动版本的 HelloLoader。由此可见，当判断类是否需要加载时，是从底层的应用类加载器开始判断的，如果已经在应用类加载器中的类，就不会请求上层类加载器了。

【示例 10-16】更进一步地修改 FindClassOrder 如下所示，并应用参数“-Xbootclasspath/a:D:/tmp/clz”执行。

```
public static void main(String args[]) throws Exception {
    ClassLoader cl=FindClassOrder3.class.getClassLoader();
    byte[] bHelloLoader=loadClassBytes("geym.zbase.ch10.findorder.HelloLoader");
    Method md_defineClass=ClassLoader.class.getDeclaredMethod("defineClass",
byte[].class,int.class,int.class);
    md_defineClass.setAccessible(true);
    md_defineClass.invoke(cl, bHelloLoader,0,bHelloLoader.length);
    md_defineClass.setAccessible(false);


    Object loader =  cl.getParent().loadClass("geym.zbase.ch10.findorder.
HelloLoader").newInstance();
```

```
11     System.out.println(loader.getClass().getClassLoader());
12     Method m=loader.getClass().getMethod("print", null);
13     m.invoke(loader, null);
14 }
```

在这段代码中，第 10 行使用扩展类加载器去尝试加载 HelloLoader，并通过反射调用 print() 方法，得到如下输出：

```
null
I am in Boot ClassLoader
```

其中，null 表示启动类加载器，最后的输出显示，从扩展类加载器中加载的 HelloLoader 是启动版本的，虽然在扩展类加载 HelloLoader 之前，该类已经在应用类加载器中了，但是扩展类加载并不会向应用类加载器进行确认，而是只在自己的路径中查找，并最终委托给了启动类加载器，而非应用类加载器，从这里可以看到，在判断类是否已经被加载时，顶层类加载器不会询问底层类加载器。

注意：判断类是否加载时，应用类加载器会顺着双亲路径往上判断，直到启动类加载器。但是启动类加载器不会往下询问，这个委托路线是单向的，理解这点很重要。

10.2.4　双亲委托模式的弊端

在前文中已经提到，检查类是否已经加载的委托过程是单向的。这种方式虽然从结构上说比较清晰，使各个 ClassLoader 的职责非常明确，但是同时会带来一个问题，即顶层的 ClassLoader 无法访问底层的 ClassLoader 所加载的类，如图 10.8 所示。

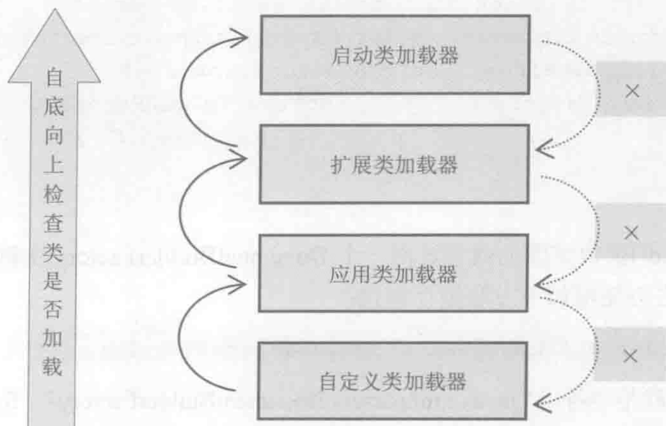


图 10.8　类加载检查顺序

通常情况下，启动类加载器中的类为系统核心类，包括一些重要的系统接口，而在应用类加载器中，为应用类。按照这种模式，应用类访问系统类自然是没有问题，但是系统类访问应用类就会出现问题。比如，在系统类中，提供了一个接口，该接口需要在应用中得以实现，该接口还绑定一个工厂方法，用于创建该接口的实例，而接口和工厂方法都在启动类加载器中。这时，就会出现该工厂方法无法创建由应用类加载器加载的应用实例的问题。拥有这种问题的组件有很多，比如 JDBC、Xml Parser 等。

10.2.5 双亲委托模式的补充

在 Java 平台中，把核心类（rt.jar）中提供外部服务，可由应用层自行实现的接口，通常可以称为 Service Provider Interface，即 SPI。

【示例 10-17】下面以 javax.xml.parsers 中实现 XML 文件解析功能模块为例，说明如何在启动类加载器中，访问由应用类加载器实现的 SPI 接口实例。

在 javax.xml.parsers.DocumentBuilderFactory 中有如下实现，用来构造一个 DocumentBuilderFactory 实例，注意 DocumentBuilderFactory 是一个抽象类（加载在启动类加载器中），可以由应用程序自行实现，这里也将介绍该方法如何返回一个在应用类加载器中的实例。

```
public static DocumentBuilderFactory newInstance() {
    try {
        return (DocumentBuilderFactory) FactoryFinder.find(
            /* The default property name according to the JAXP spec */
            "javax.xml.parsers.DocumentBuilderFactory",
            /* The fallback implementation class name */
            "com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl");
    } catch (FactoryFinder.ConfigurationError e) {
        throw new FactoryConfigurationError(e.getException(),
                                            e.getMessage());
    }
}
```

FactoryFind.find()函数试图加载并返回一个 DocumentBuilderFactory 实例。当这个实例在应用层 jar 包里时，它会使用如下方法进行查找：

```
Object provider = findJarServiceProvider(factoryId);
```

其中 factoryId 就是字符串“javax.xml.parsers.DocumentBuilderFactory”，findJarServiceProvider 的主要内容如下代码所示，这段代码并非 JDK 中的源码，为了节省版面，笔者做了适当的裁剪，只保留核心部分。

```
private static Object findJarServiceProvider(String factoryId)
    throws ConfigurationError
{
    String serviceId = "META-INF/services/" + factoryId;
    InputStream is = null;
    ClassLoader cl = ss.getContextClassLoader();
    InputStream is = ss.getResourceAsStream(cl, serviceId);
    BufferedReader rd = new BufferedReader(new InputStreamReader(is, "UTF-8"));
    String factoryClassName = rd.readLine();
    return newInstance(factoryClassName, cl, false, useBSClsLoader);
}
```

从以上代码可知，系统通过读取 jar 包中 META-INF/services 目录下的类名文件，读取工厂类类名，然后根据类名生成对应的实例。加粗部分是本段代码关键，这里获得了一个名为上下文加载器的 ClassLoader（加粗部分），并将此 ClassLoader 传入 newInstance()方法，由这个 ClassLoader 去完成实例的加载和创建，而不是由这段代码所在的启动类加载器去加载。从而解决了启动类加载器无法访问 factoryClassName 指定类的问题。

最后，再看一下 ss.getContextClassLoader()是如何获得这个上下文加载器的，从以下代码可知，上下文加载器是从 Thread. getContextClassLoader()中得到的。

```
ClassLoader getContextClassLoader() throws SecurityException{
    return (ClassLoader)
            AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() {
            ClassLoader cl = null;
            //try {
            cl = Thread.currentThread().getContextClassLoader();
            //} catch (SecurityException ex) { }

            if (cl == null)
                cl = ClassLoader.getSystemClassLoader();

            return cl;
        }
    });
}
```

查看 Thread 类的两个方法：

```
public ClassLoader getContextClassLoader() {
public void setContextClassLoader(ClassLoader cl)
```

这两个方法分别是取得设置在线程中的上下文加载器和设置一个线程的上下文加载器。通过这两个方法，可以把一个 ClassLoader 置于一个线程实例之中，使该 ClassLoader 成为一个相对共享的实例。默认情况下，上下文加载器就是应用类加载器，这样即使是在启动类加载器中的代码也可以通过这种方式访问应用类加载器中的类了。图 10.9 显示了以上下文加载器作为中介，使得启动类加载器得以访问应用类加载器的类。

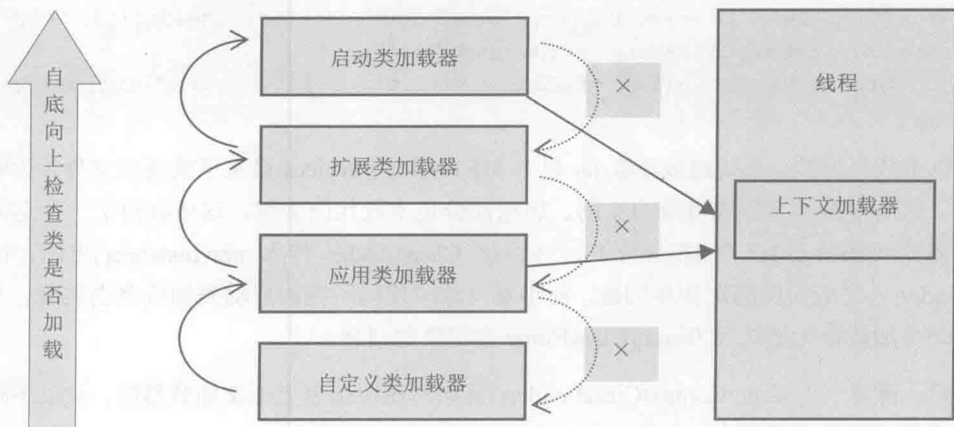


图 10.9　上下文加载器示意图

10.2.6　突破双亲模式

双亲模式的类加载方式是虚拟机默认的行为，但并非必须这么做，通过重载 ClassLoader 可以修改该行为。事实上，不少应用软件和框架都修改了这种行为，比如 Tomcat 和 OSGi 框架，都有各自独特的类加载顺序。在本小节中，将演示如何打破默认的双亲模式。

【示例 10-18】下面的代码通过重载 loadClass()方法，改变类的加载次序，这里给出部分核心代码：

```
public class OrderClassLoader extends ClassLoader {
    private String fileName;

    public OrderClassLoader(String fileName) {
        this.fileName = fileName;
    }

     protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {
```

```
        // First, check if the class has already been loaded
        Class re = findClass(name);
        if (re == null) {
            System.out.println("I can't load the class:" + name + " need help from parent");
            return super.loadClass(name, resolve);
        }
        return re;
    }

    protected Class<?> findClass(String className) throws ClassNotFoundException {
        Class clazz = this.findLoadedClass(className);
        if (null == clazz) {
            try {
                String classFile = getClassFile(className);
                FileInputStream fis = new FileInputStream(classFile);
                FileChannel fileC = fis.getChannel();
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                WritableByteChannel outC = Channels.newChannel(baos);
                ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
                while (true) {
                    int i = fileC.read(buffer);
                    if (i == 0 || i == -1) {
                        break;
                    }
                    buffer.flip();
                    outC.write(buffer);
                    buffer.clear();
                }
                fis.close();
                byte[] bytes = baos.toByteArray();
                clazz = defineClass(className, bytes, 0, bytes.length);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return clazz;
    }
此处省略部分非核心代码
```

以上代码通过自定义 ClassLoader，重载 loadClass()改变了默认的委托双亲加载的方式。第 10 行通过 findClass()读取 class 文件，并将二进制流定义为 Class 对象。如果加载不到，则委托双亲加载，这种方式颠倒了默认的加载顺序。

下面的 ClassLoaderTest 使用上述的 OrderClassLoader 进行工作。

```
public class ClassLoaderTest {
    public static void main(String[] args) throws ClassNotFoundException {
        OrderClassLoader myLoader=new OrderClassLoader("D:/tmp/clz/");
        Class clz=myLoader.loadClass("geym.zbase.ch10.brkparent.DemoA");
        System.out.println(clz.getClassLoader());

        System.out.println("==== Class Loader Tree ====");
        ClassLoader cl=clz.getClassLoader();
        while(cl!=null){
            System.out.println(cl);
            cl=cl.getParent();
        }
    }
}
```

以上代码创建了 OrderClassLoader 对象，将“D:/tmp/clz/”目录作为其类的搜索路径，然后使用 OrderClassLoader 加载 DemoA 类（DemoA 亦类在当前 ClassPath 中），但是通过程序的输出可以看到，实际加载 DemoA 的是 OrderClassLoader（即使它的双亲应用类加载器也能加载到该类）。运行以上代码，输出如下：

```
java.io.FileNotFoundException: D:\tmp\clz\java\lang\Object.class (系统找不到指定的路径。)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:146)
    at java.io.FileInputStream.<init>(FileInputStream.java:101)
    at geym.zbase.ch10.brkparent.OrderClassLoader.findClass(OrderClassLoader.java:41)
    at geym.zbase.ch10.brkparent.OrderClassLoader.loadClass(OrderClassLoader.java:28)
    省略部分输出
    at geym.zbase.ch10.brkparent.OrderClassLoader.loadClass(OrderClassLoader.java:28)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at geym.zbase.ch10.brkparent.ClassLoaderTest.main(ClassLoaderTest.java:12)
I can't load the class:java.lang.Object need help from parent
```

```
geym.zbase.ch10.brkparent.OrderClassLoader@1833eca
==== Class Loader Tree ====
geym.zbase.ch10.brkparent.OrderClassLoader@1833eca
sun.misc.Launcher$AppClassLoader@b23210
sun.misc.Launcher$ExtClassLoader@f4f44a
```

可以到，程序首先试图由 OrderClassLoader 加载 Object 类，但由于“D:/tmp/clz/”路径中没有该类信息，故加载失败，抛出异常，但随后就由应用类加载器加载成功，接着尝试加载 DemoA，OrderClassLoader 顺利在“D:/tmp/clz/”中找到该类信息，并且加载成功（该实验中 DemoA 亦在 ClassPath 中，可以由应用类加载器加载），打印加载 DemoA 类的 ClassLoader，显示为 OrderClassLoader，打印 ClassLoader 的层次，依次为 OrderClassLoader、AppClassLoader、ExtClassLoader。

10.2.7　热替换的实现

热替换是指在程序的运行过程中，不停止服务，只通过替换程序文件来修改程序的行为。热替换的关键需求在于服务不能中断，修改必须立即表现在正在运行的系统之中。基本上大部分脚本语言都是天生支持热替换的，比如 PHP，只要替换了 PHP 源文件，这种改动就会立即生效，而无须重启 Web 服务器。

但对 Java 来说，热替换并非天生就支持，如果一个类已经加载到系统中，通过修改类文件，并无法让系统再来加载并重定义这个类。因此，在 Java 中实现这一功能的一个可行的方法就是灵活运用 ClassLoader。

说明： 在这里，希望读者首先理解一个概念：由不同 ClassLoader 加载的同名类属于不同的类型，不能相互转化和兼容。

【示例 10-19】来看一个例子，在路径“D:/tmp/clz”下存放一个类“geym.zbase.ch10.findorder.HelloLoader”，同时在当前 ClassPath 下也存放同名类“geym.zbase.ch10.findorder.HelloLoader”，前者会打印字符串“I am in Boot ClassLoader”，后者会打印字符串“I am in apploader”。

使用参数“-Xbootclasspath/a:D:/tmp/clz”运行以下代码：

```
public static void main(String args[]) throws Exception {
    ClassLoader cl = SameNameClass.class.getClassLoader();
    HelloLoader loader = (HelloLoader) cl.getParent().loadClass("geym.zbase.
ch10.findorder.HelloLoader")
            .newInstance();
    System.out.println(loader.getClass().getClassLoader());
```

```
    loader.print();
}
```

会输出：

```
null
I am in Boot ClassLoader
```

正如前文所描述的，HelloLoader 由启动类加载器加载。现将代码修改如下：

```
01 public static void main(String args[]) throws Exception {
02     ClassLoader cl = SameNameClass.class.getClassLoader();
03     byte[] bHelloLoader = loadClassBytes("geym.zbase.ch10.findorder.HelloLoader");
04          Method md_defineClass = ClassLoader.class.getDeclaredMethod
("defineClass", byte[].class, int.class, int.class);
05     md_defineClass.setAccessible(true);
06     md_defineClass.invoke(cl, bHelloLoader, 0, bHelloLoader.length);
07     md_defineClass.setAccessible(false);
08
09     HelloLoader loader = (HelloLoader) cl.getParent().loadClass("geym.
zbase.ch10.findorder.HelloLoader")
10             .newInstance();
11     System.out.println(loader.getClass().getClassLoader());
12     loader.print();
13 }
```

在第 3~7 行，将 HelloLoader 加载到应用类加载器中，继而使用启动类加载器再加载一次 HelloLoader，此时，系统内含有两个同名的 HelloLoader 类型，在第 9 行做了强制转换，运行以上代码抛出如下错误：

```
Exception in thread "main" java.lang.ClassCastException: geym.zbase.ch10.
findorder.HelloLoader cannot be cast to geym.zbase.ch10.findorder.HelloLoader
    at geym.zbase.ch10.clshot.SameNameClass.main(SameNameClass.java:32)
```

从这个错误可以看到，两个同名类居然无法相互转换，这是因为它们由不同的 ClassLoader 加载的。

注意： 两个不同 ClassLoader 加载同一个类，在虚拟机内部，会认为这 2 个类是完全不同的。

【示例 10-20】使用这个特点，可以用来模拟热替换的实现，基本思路如图 10.10 所示。

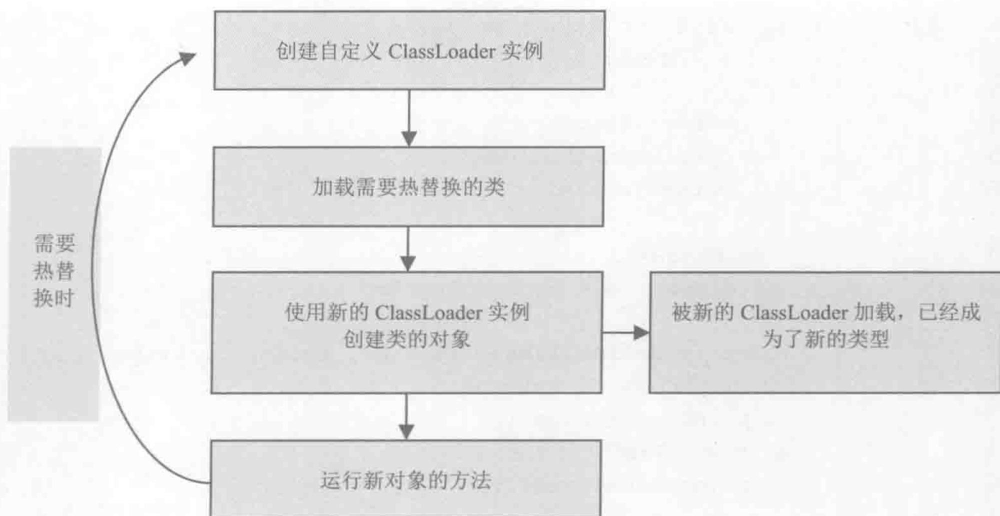


图 10.10 热替换基本思路

在实现时，首先需要自定义 ClassLoader，以下代码自定义了 ClassLoader，它可以在给定目录下查找目标类，主要的实现思路是重载 findClass()方法。

```
public class MyClassLoader extends ClassLoader {
    private String fileName;

    public MyClassLoader(String fileName) {
        this.fileName = fileName;
    }

    protected Class<?> findClass(String className) throws ClassNotFoundException {
        Class clazz = this.findLoadedClass(className);
        if (null == clazz) {
            try {
                String classFile = getClassFile(className);
                FileInputStream fis = new FileInputStream(classFile);
                FileChannel fileC = fis.getChannel();
                ByteArrayOutputStream baos = new
                        ByteArrayOutputStream();
                WritableByteChannel outC = Channels.newChannel(baos);
                ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
                while (true) {
                    int i = fileC.read(buffer);
```

```
21                  if (i == 0 || i == -1) {
22                      break;
23                  }
24                  buffer.flip();
25                  outC.write(buffer);
26                  buffer.clear();
27              }
28              fis.close();
29              byte[] bytes = baos.toByteArray();
30
31              clazz = defineClass(className, bytes, 0, bytes.length);
32          } catch (FileNotFoundException e) {
33              e.printStackTrace();
34          } catch (IOException e) {
35              e.printStackTrace();
36          }
37      }
38      return clazz;
39  }
40  以下省略非核心代码
```

在 findClass()的实现中，在第 9 行，查找已经加载的类，如果类已经加载，则不作重复加载。第 11~29 行，通过文件查找，读取 Class 的二进制数据流。第 31 行，将此二进制数据流定义为 Class，并返回该 Class 对象。

准备一个需要被热替换的类，命名为 DemoA，代码如下，调用其 hot()方法，将打印字符串“OldDemoA”。

```
public class DemoA {
    public void hot(){
        System.out.println("OldDemoA");
    }
}
```

将生成的 DemoA 的 class 文件放置于目录 D:\tmp\clz 下。建立热替换支持类 DoopRun，它使用 MyClassLoader 在路径 D:/tmp/clz 下查找并且更新 DemoA 的实现。

```
public class DoopRun {
    public static void main(String args[]) {
        while(true){
            try{
                MyClassLoader loader = new MyClassLoader("D:/tmp/clz");
```

```
                Class cls = loader.loadClass("geym.zbase.ch10.clshot.DemoA");
                Object demo = cls.newInstance();
                Method m = demo.getClass().getMethod("hot", new Class[] {});
                m.invoke(demo, new Object[] {});
                Thread.sleep(10000);
            }catch(Exception e){
                System.out.println("not find");
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException e1) {
                }
            }
        }
    }
}
```

以上代码每次在调用 DemoA.hot()方法之前，都会重新加载 DemoA，从而实现热替换。运行以上代码，程序将不停输出“OldDemoA”字符串。

修改 DemoA.hot()的输出字符为“NewDemoA”，重新生成新的 class 文件，并将其覆盖到 D:/tmp/clz 下，在不停止 DoopRun 程序的情况下发现 DemoA 可以被更新：

```
OldDemoA
NewDemoA
NewDemoA
```

10.3　小结

本章主要介绍了 Java 虚拟机的类装载系统。首先介绍了 Class 文件装载的全过程和每个阶段的主要工作。接着，着重介绍了 ClassLoader 的工作机制，双亲模式的工作过程以及打破双亲模式的方法。最后，基于 ClassLoader 的原理，给出了一个热替换的方案。了解 ClassLoader，有助于读者在工作中更好地理解现有框架的实现。

第 11 章 字节码执行

字节码执行是 Java 虚拟机的重点，就如同汇编语言对于计算机一样重要，字节码对于 Java 虚拟机来说是执行的根本。当 Java 源码被编译成 Class 文件后，虚拟机就会将 Class 文件内的方法字节码载入系统并加以执行。本章将详细介绍虚拟机的指令系统、执行机制以及相关配置参数。

本章涉及的主要知识点有：

- 使用 javap 查看 Class 信息。
- 了解字节码执行过程。
- 熟悉常用字节码。
- 学习 JIT 相关参数配置。
- 通过 ASM 增强方法功能。
- 使用 Java Agent 动态修改 Class 字节码。

11.1 代码如何执行：字节码执行案例

Java 字节码对于虚拟机，就好像汇编语言对于计算机，属于基本执行指令。每一个 Java 字节码指令是一个 byte 数字，并且有一个对应的助记符。目前所有的字节码指令大约有 200 余个。下面列举了部分字节码及其对应的助记符：

```
_nop                  =   0,       // 0x00
_aconst_null          =   1,       // 0x01
_iconst_0             =   3,       // 0x03
_iconst_1             =   4,       // 0x04
_dconst_1             =  15,       // 0x0f
_bipush               =  16,       // 0x10
_iload_0              =  26,       // 0x1a
_iload_1              =  27,       // 0x1b
_aload_0              =  42,       // 0x2a
_istore               =  54,       // 0x36
_pop                  =  87,       // 0x57
_imul                 = 104,       // 0x68
_idiv                 = 108,       // 0x6c
```

一个方法的 Java 字节码指令，被编译到 Java 方法的 Code 属性中，如果想要查看指令的具体内容，可以使用 JDK 自带的 javap 工具。javap 的常用参数如下：

```
用法：javap <options> <classes>
其中，可能的选项包括：
  -version                  版本信息
  -v  -verbose              输出附加信息
  -l                        输出行号和本地变量表
  -public                   仅显示公共类和成员
  -protected                显示受保护的/公共类和成员
  -package                  显示程序包/受保护的/公共类和成员（默认）
  -p  -private              显示所有类和成员
  -c                        对代码进行反汇编
  -s                        输出内部类型签名
  -sysinfo                  显示正在处理的类的系统信息（路径、大小、日期、MD5 散列）
  -constants                显示静态最终常量
```

【示例 11-1】下面以一段简单的 Java 代码为例，演示 javap 的使用。

```
package geym.zbase.ch11.calc;
public class Calc {
   public int calc() {
```

```
        int a = 500;
        int b = 200;
        int c = 50;
        return (a + b) / c;
    }
}
```

用 javap 显示这段代码的信息，命令和返回信息如下：

```
javap -v geym\zbase\ch11\calc\Calc
Compiled from "Calc.java"
public class geym.zbase.ch11.calc.Calc extends java.lang.Objec
  SourceFile: "Calc.java"
  minor version: 0
  major version: 51
  Constant pool:
const #1 = class        #2;     //  geym/zbase/ch11/calc/Calc
const #2 = Asciz        geym/zbase/ch11/calc/Calc;
const #3 = class        #4;     //  java/lang/Object
const #4 = Asciz        java/lang/Object;
const #5 = Asciz        <init>;
const #6 = Asciz        ()V;
const #7 = Asciz        Code;
const #8 = Method       #3.#9;  //  java/lang/Object."<init>":
const #9 = NameAndType  #5:#6;//  "<init>":()V
const #10 = Asciz       LineNumberTable;
const #11 = Asciz       LocalVariableTable;
const #12 = Asciz       this;
const #13 = Asciz       Lgeym/zbase/ch11/calc/Calc;;
const #14 = Asciz       calc;
const #15 = Asciz       ()I;
const #16 = Asciz       a;
const #17 = Asciz       I;
const #18 = Asciz       b;
const #19 = Asciz       c;
const #20 = Asciz       SourceFile;
const #21 = Asciz       Calc.java;

{
public geym.zbase.ch11.calc.Calc();
  Code:
   Stack=1, Locals=1, Args_size=1
```

```
    0:   aload_0
    1:   invokespecial   #8; //Method java/lang/Object."<init>"
    4:   return
  LineNumberTable:
   line 3: 0

  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      5       0    this      Lgeym/zbase/ch11/calc/Calc;


public int calc();
  Code:
   Stack=2, Locals=4, Args_size=1
   0:   sipush  500
   3:   istore_1
   4:   sipush  200
   7:   istore_2
   8:   bipush  50
   10:  istore_3
   11:  iload_1
   12:  iload_2
   13:  iadd
   14:  iload_3
   15:  idiv
   16:  ireturn
  LineNumberTable:
   line 5: 0
   line 6: 4
   line 7: 8
   line 8: 11

  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      17      0     this      Lgeym/zbase/ch11/calc/Calc;
   4      13      1     a      I
   8      9      2     b      I
   11      6      3     c      I
}
```

从上述代码可以看出，这段简单的 Java 程序被反编译后生成了大量的信息。

首先，显示了生成这个 Class 文件的 Java 源文件名称、小版本和大版本号。接着，显示了这个类中所有的常量，在这里有 21 项常量。此外，还显示了两个方法信息：第 1 个方法为类的构造函数，是编译器自动插入的；第 2 个方法为 calc()方法，在方法体内，显示了栈大小、局部变量表大小、字节码指令、行号、局部变量表等信息。

读者可以重点关注一下 calc()方法的主体内容，即字节码部分（加粗代码），下面将仔细解读这段字节码的执行过程。如图 11.1 所示，左侧的字节码列表用加粗字体显示了正在执行的字节码，右侧分别显示了当前的局部变量表和操作数栈。在字节码部分，左侧的数字序号表示字节码偏移量，即当前字节码所在的位置，这看起来有点像行号，但它不是。字节码偏移量总是和前几个字节码的长度有关，第一条字节码为 sipush，自然其偏移量为 0。而 sipush 这条指令本身占用 1 个字节，但它接收一个双字节的参数，故整个 sipush 指令合计 3 个字节码，因此，其后续指令 istore_1 所在位置在偏移量 3 处，而 istore_1 指令为 1 字节，且不接收参数，故合计 1 字节，加上之前 sipush 的 3 字节，sipush 200 就在偏移量 4 的位置，以此类推。

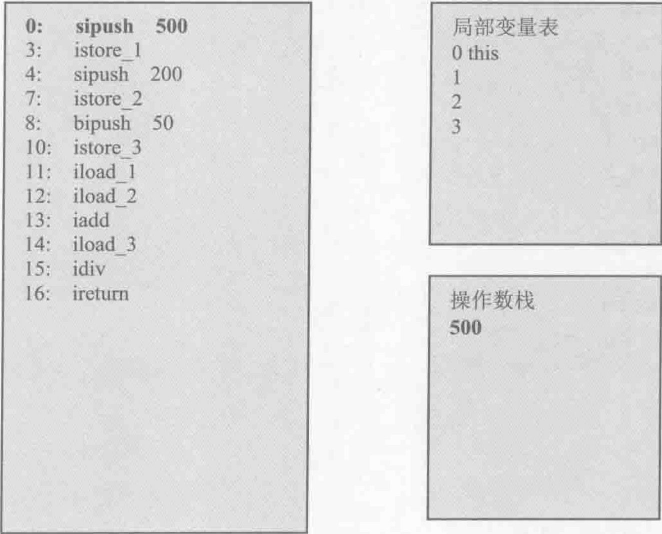


图 11.1　字节码执行示意图 1

如图 11.1 所示，在执行第一条指令的时候。局部变量表第 0 项为 this 引用，表示当前对象。在 Java 中，对于所有的非静态函数调用，为了能顺利访问 this 对象，都会将对象的引用放置在局部变量表第 0 个槽位。指令 sipush 的作用是将给定的参数压入操作数栈，故执行完 sipush 500 后，操作数栈中含有数字 500。在虚拟机的指令集中，还有一条指令为 bipush，也是完成相同的功能，但是 bipush 仅接收一个字节作为其参数，因此，它只能处理-128~127 的数字范围，这

里的 500 已经超过了 bipush 的处理范围，故使用 sipush，它可以支持-32768 ~32767。虚拟机通过这种细分的指令集，可以尽可能减少指令所占的空间，毕竟 sipush 要比 bipush 多占一个字节。

如图 11.2 所示，虚拟机正在执行 istore_1 命令，store 命令是从操作数栈中弹出一个元素，并将其存放在局部变量表中。一般说来，类似像 store 这样的命令需要带一个参数，用来指明将弹出的元素放在局部变量表的第几个位置。但是，为了尽可能压缩指令大小，使用专门的 istore_1 指令表示将弹出的元素放置在局部变量表第 1 个位置。类似的还有 istore_0、istore_2、istore_3，它们分别表示从操作数栈顶弹出一个元素，存放在局部变量表第 0、2、3 个位置。由于局部变量表前几个位置总是非常常用，因此这种做法虽然增加了指令数量，但是可以大大压缩生成的字节码的体积。如果局部变量表很大，需要存储的槽位大于 3，那么可以使用 istore 指令，外加一个参数，用来表示需要存放的槽位位置。

故在这条指令执行完毕后，操作数栈被清空，局部变量表 1 的位置存入 500。

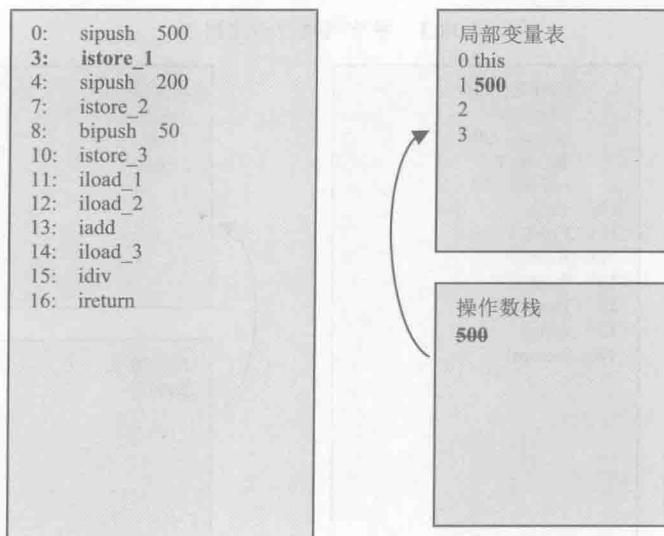


图 11.2　字节码执行示意图 2

接着，执行 sipush 200，将 200 压入操作数栈，如图 11.3 所示。

指令 istore2，将 200 存入局部变量第 2 个位置，并清空操作数栈。如图 11.4 所示。

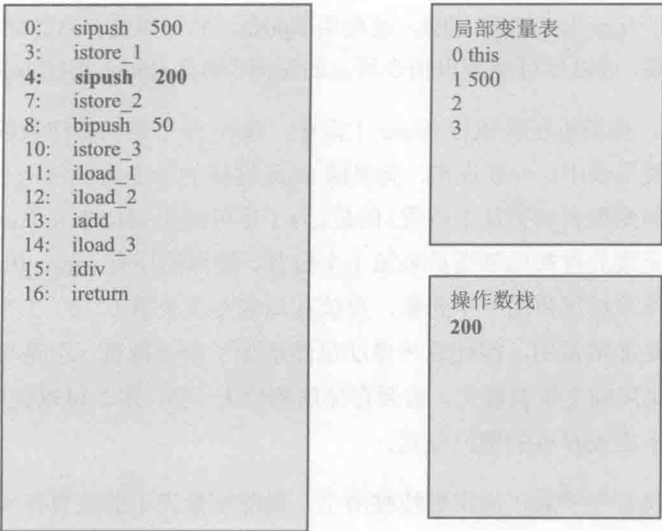


图 11.3 字节码执行示意图 3

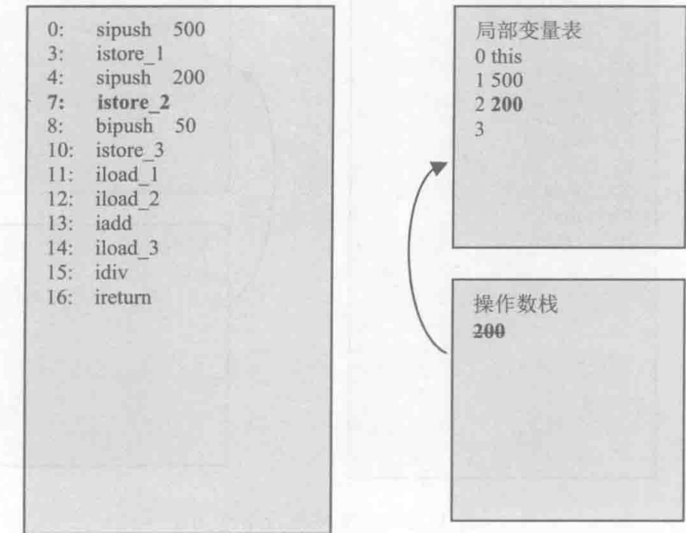


图 11.4 字节码执行示意图 4

指令 bipush 50 将 50 压入操作数栈，如图 11.5 所示。

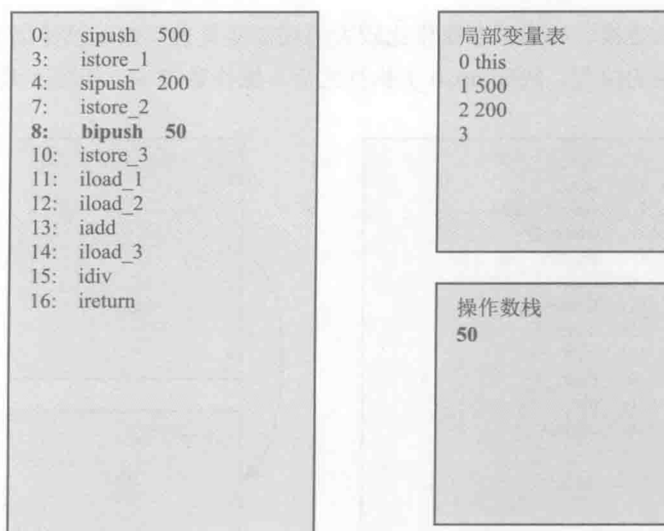


图 11.5　字节码执行示意图 5

指令 istore_3 将 50 弹出，并存放在局部变量表第 3 个位置，如图 11.6 所示。

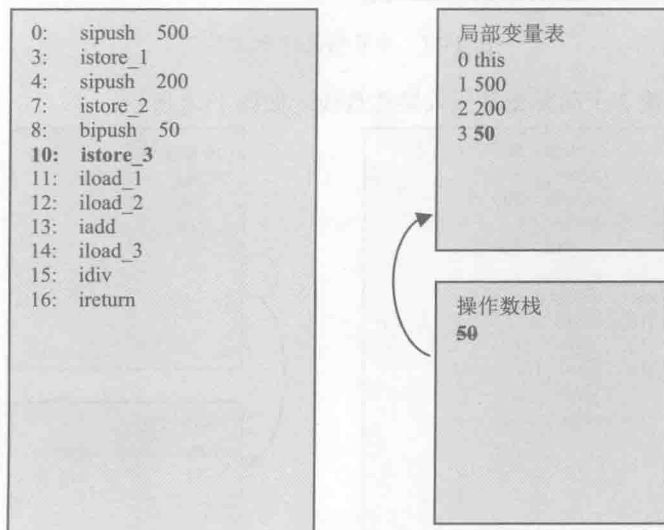


图 11.6　字节码执行示意图 6

指令 iload_1 将局部变量表第 1 个位置的值压入操作数栈。和 store 指令类似，虚拟机也提供了一系列类似的指令，比如 iload_0、iload_2、iload_3 等，分别表示将局部变量表第 0、2、3

个位置的值压入操作数栈。如果需要操作比较大的局部变量表，则可以使用 iload，外加一个参数来指明局部变量表的位置。故在 iload_1 执行之后，操作数栈中，栈顶元素为 500，如图 11.7 所示。

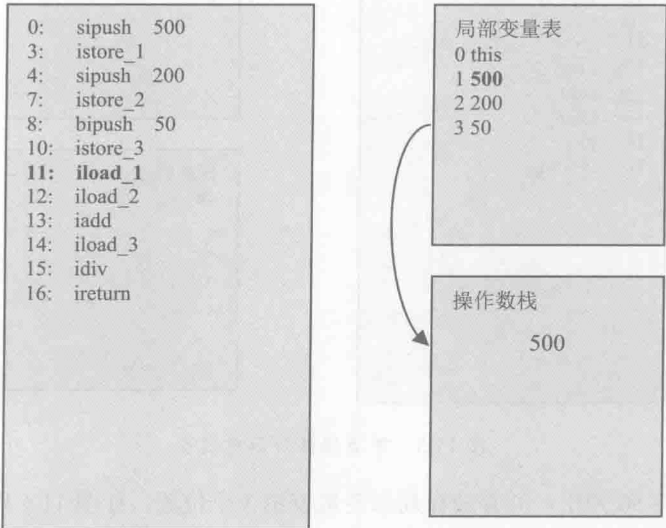


图 11.7　字节码执行示意图 7

指令 iload_2 将第 2 个局部变量压入操作数栈，如图 11.8 所示。

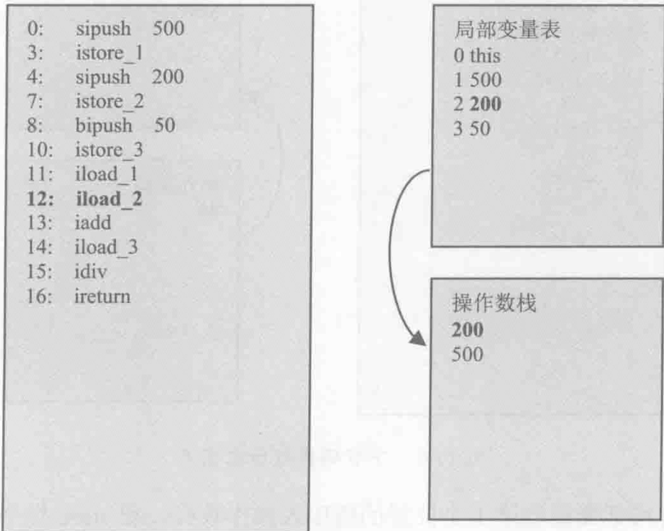


图 11.8　字节码执行示意图 8

指令 iadd 表示加法操作，它从操作数栈中弹出两个元素做加法，并将结果再压回操作数栈，如图 11.9 所示。

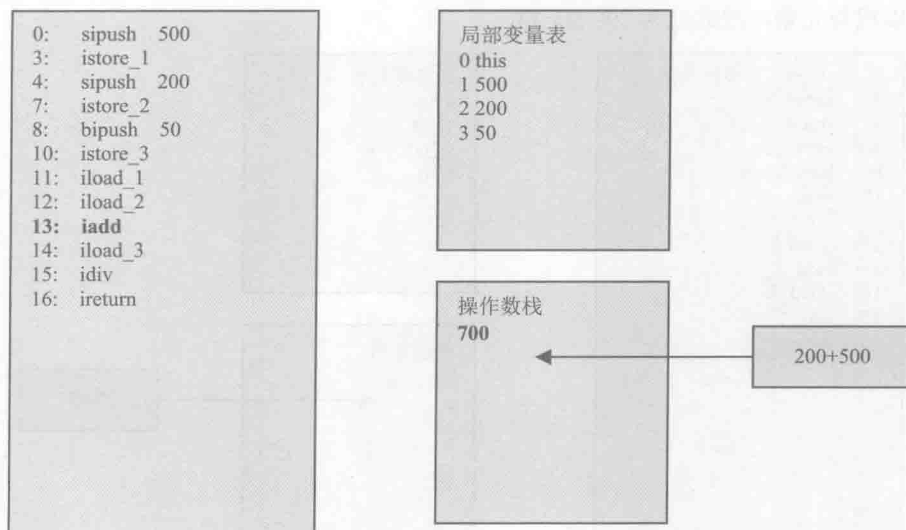


图 11.9　字节码执行示意图 9

指令 iload_3 将局部变量表第 3 位的 50 压入操作数栈，结果如图 11.10 所示。

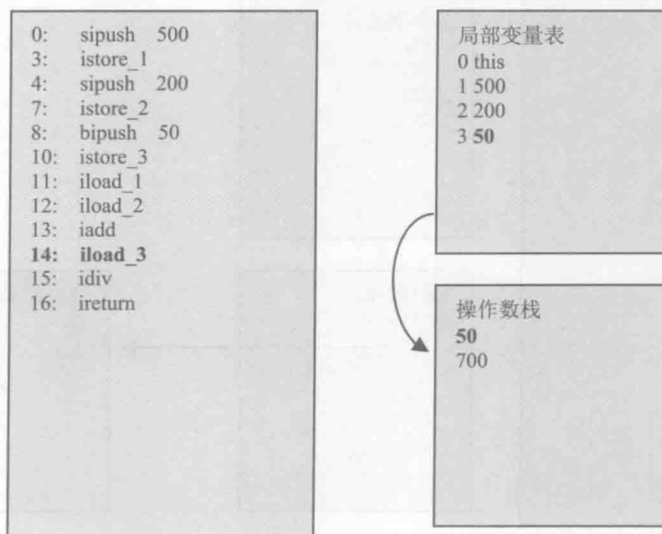


图 11.10　字节码执行示意图 10

指令 idiv 表示整数除法，它从操作数栈中弹出两个元素，相除后将结果压入操作数栈，如图 11.11 所示。比 iadd 略显复杂的是，除法是需要考虑顺序的，idiv 的做法是用栈顶第 2 顺位的元素除以栈顶元素，故此处为 700/50=14。

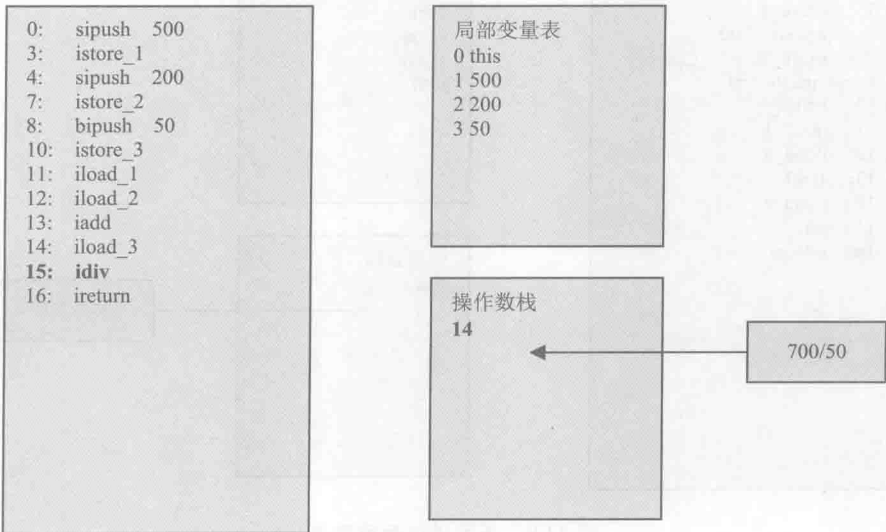


图 11.11　字节码执行示意图 11

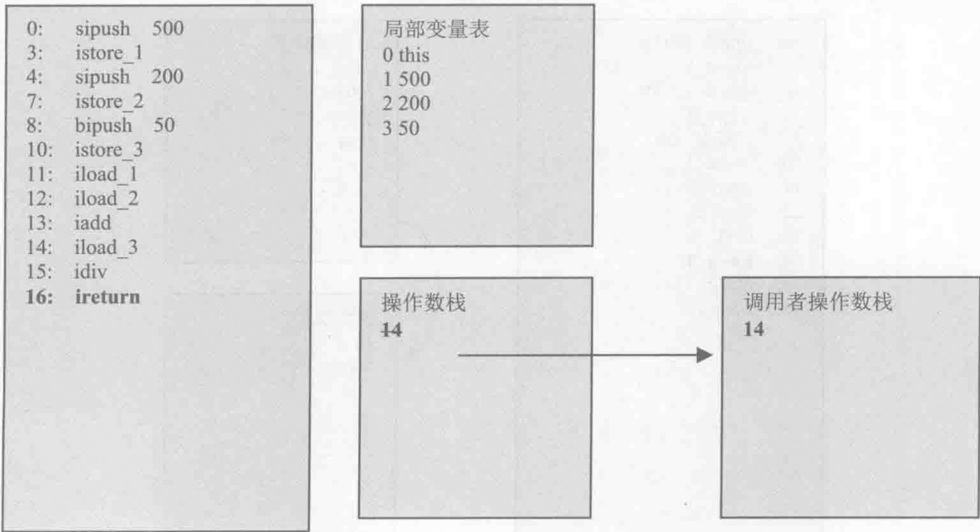


图 11.12　字节码执行示意图 12

在最后，通过 ireturn 指令（图 11.12），将当前函数操作数栈的顶层元素弹出，并将这个元素压入调用者函数的操作数栈中（因为调用者非常关心函数的返回值），所有在当前函数操作数栈中的其他元素都会被丢弃。如果当前返回是 synchroinzed 方法，那么还会执行一个隐含的 monitorexit 指令退出临界区。最后，会丢弃当前方法的整个帧，恢复调用者的帧，并将控制权转交给调用者。

11.2　执行的基础：Java 虚拟机常用指令介绍

Java 虚拟机的字节码指令多达 200 余个，完全介绍和学习这些指令需要花费大量时间。为了让读者能够更快地熟悉和了解这些基本指令，本节对常用的指令做了归类整理，方便读者查阅。熟悉虚拟机的指令对于动态字节码生成、反编译 Class 文件、Class 文件修补都有着非常重要的价值。

11.2.1　常量入栈指令

常量入栈指令的功能是将常数压入操作数栈，根据数据类型和入栈内容的不同，又可以分为 const 系列、push 系列和 ldc 指令。

指令 const 系列用于对特定的常量入栈，入栈的常量隐含在指令本身里。比如，aconst_null 将 null 压入操作数栈；iconst_m1 将-1 压入操作数栈；iconst_x（x 为 0 到 5）将 x 压入栈；lconst_0、lconst_1 分别将长整数 0 和 1 压入栈；fconst_0、fconst_1、fconst_2 分别将浮点数 0、1、2 压入栈；dconst_0 和 dconst_1 分别将 double 型 0 和 1 压入栈。从指令的命名上，不难找出规律，指令助记符的第一个字符总是喜欢表示数据类型，i 表示整数，l 表示长整数，f 表示浮点数，d 表示双精度浮点，习惯上用 a 表示对象引用。如果指令隐含操作的参数，会以下划线形式给出。

指令 push 系统主要包括 bipush 和 sipush。它们的区别在于接收数据类型的不同，bipush 接收 8 位整数作为参数，sipush 接收 16 位整数，它们都将参数压入栈。

如果以上指令都不能满足需求，那么可以使用万能的 ldc 指令，它可以接收一个 8 位的参数，该参数指向常量池中的 int、float 或者 String 的索引，将制定的内容压入堆栈。类似的还有 ldc_w，它接收两个 8 位参数，能支持的索引范围大于 ldc。如果要压入的元素是 long 或者 double，则使用 ldw2_w 指令，使用方式都是类似的。

以下代码使用了常量 100000.0d：

```
System.out.println(100000.0d);
```

它会产生的指令中就包含了 ldc2w。如下所示，第 2 行为常量池第 27 项。

```
ldc2_w      #27
#27 = Double          100000.0d
```

11.2.2 局部变量压栈指令

局部变量压栈指令将给定的局部变量表中的数据压入操作数栈。这类指令大体可以分为：xload（x 为 i、l、f、d、a）、xload_n（x 为 i、l、f、d、a，n 为 0 到 3）、xaload（x 为 i、l、f、d、a、b、c、s）。

在这里，x 的取值表示数据类型，其值如表 11.1 所示。

表 11.1　x所表示的数据类型

x取值	含　义
i	int整数
l	长整数
f	浮点数
d	双精度浮点
a	对象索引
b	byte
c	char
s	short

指令 xload_n，表示将第 n 个局部变量压入操作数栈，比如 iload_1、fload_0、aload_0 等指令。其中 aload_n 表示将一个对象引用压栈。

指令 xload 通过指定参数的形式，把局部变量压入操作数栈，当使用这个命令时，表示局部变量的数量可能超过了 4 个，比如指令 iload、fload 等。

指令 xaload 表示将数组的元素压栈，比如 saload、caload 分别表示压入 short 数组和 char 数组。指令 xaload 在执行时，要求操作数中栈顶元素为数组索引 i，栈顶顺位第 2 个元素为数组引用 a，该指令会弹出栈顶这两个元素，并将 a[i]重新压入堆栈。

【示例 11-2】以下面的代码为例。

```
public void print3(char[] cs,short[] s){
    System.out.println(s[0]);
    System.out.println(cs[0]);
}
```

生成的部分字节码如下：

```
0: getstatic     #21
3: aload_2
4: iconst_0
5: saload
6: invokevirtual #43
9: getstatic     #21
12: aload_1
13: iconst_0
14: caload
```

注意加粗部分，符合前文的描述，在 saload 执行前，先将数组引用入栈（aload_2），在将索引入栈（iconst_0），索引为 0。同理在 caload 执行前，先将 char[]数组压栈（aload_1），然后将数组索引 0 压栈（iconst_0），最后调用 caload 将第 0 位的数据压入操作数栈顶部。

11.2.3　出栈装入局部变量表指令

出栈装入局部变量表指令用于将操作数栈中栈顶元素弹出后，装入局部变量表的指定位置，用于给局部变量赋值。这类指令主要以 store 的形式存在，比如 xstore（x 为 i、l、f、d、a）、xstore_n（x 为 i、l、f、d、a，n 为 0 到 3）和 xastore（x 为 i、l、f、d、a、b、c、s）。其中 x 的取值含义和 load 类命令是一样的，在此不再赘述。

指令 istore_1 将从操作数栈中弹出一个整数，并把它赋值给局部变量 1。指令 xstore 由于没有隐含参数信息，故需要提供一个 byte 类型的参数类指定目标局部变量表的位置。xastore 则专门针对数组操作，以 iastore 为例，它用于给一个 int 数组的给定索引赋值。在 iastore 执行前，操作数栈顶需要以此准备 3 个元素：值、索引、数组引用，iastore 会弹出这 3 个值，并将值赋给数组中指定索引的位置。

【示例 11-3】以下面这段简单的代码为例。

```
public void print4(char[] cs,int[] s){
   int i,j,k,x;
   x=99;
   s[0]=77;
}
```

它生成的字节码中包含 istore 和 iastore 指令。

```
0: bipush        99
```

```
2: istore        6
4: aload_2
5: iconst_0
6: bipush        77
8: iastore
9: return
```

其中，第 0 行字节码（实际上是字节码偏移量，为表述简单起见，使用行为单位，下同），压入常量 99，第 2 行 istore 将 99 弹出，并赋给局部变量表 6 的变量，而该变量正好是 x。接着在第 4、5、6 行分别压入 iastore 所需的 3 个参数，最后调用 iastore 将 77 赋值给局部变量表第 2 位（int[] s）数组的第 0 个索引位置（s[0]）。

11.2.4 通用型操作

从上面的几类指令中可以看到，大部分数据操作指令是和数据类型相关的。Java 虚拟机为不同的数据类型都量身定做了一系列指令。但是无类型的指令还是必要的，比如就栈操作而言，不是在所有时刻对栈的压入或者弹出都必须明确数据类型的。通用型操作就提供了这种无需指明数据类型的操作。

指令 NOP，是一个非常特殊的指令，它的字节码为 0x00。和汇编语言中的 nop 一样，它表示什么都不做。这条指令一般可用于调试、占位等。

另外两个比较重要的操作是 dup 和 pop 指令。指令 dup 意为 duplicate 复制，它会将栈顶元素复制一份并再次压入栈顶，这样栈顶就有两份一模一样的元素了。指令 pop 则把一个元素从栈顶弹出，并且直接废弃。

【示例 11-4】下面这段简单的代码是这两个指令的使用案例。

```
public void print5(int i) {
    Object obj=new Object();
    obj.toString();
}
```

编译成字节码后，其内容如下：

```
0: new            #3        // class java/lang/Object
3: dup
4: invokespecial #16       // Method java/lang/Object."<init>":()V
7: astore_2
8: aload_2
9: invokevirtual #58       // Method java/lang/Object.toString:()Ljava/lang/String;
```

```
12: pop
13: return
```

为了生成 Object 对象，使用了对象创建指令 new。创建完成后，new 指令会把对象引用放置在栈顶，此时，栈顶只有一份对象引用。但是，在 new 指令之后，对该 obj 对象连续进行两次操作：一次是通过 invokespecial 指令调用对象的构造函数，另一次是通过 astore_2 将对象赋值给 obj。这两个操作都会将栈顶元素弹出，故为了连续两次使用同样的栈顶元素，这里使用指令 dup 赋值了一份对象引用，供后面连续两次指令使用。在 obj.toString()方法执行完毕后，函数的范围值会出现在栈顶，但是由于没有人使用，故简单地使用 pop 操作将无人问津的返回值直接丢弃，也实属合理。

注意： pop 指令只能丢弃 1 个字长（32 位），如果要丢弃栈顶 64 位数据（long 或者 double），则需要使用 pop2 命令，类似地，如果要连续复制栈顶 2 个字长，则可以使用 dup2 指令。

11.2.5　类型转换指令

在软件开发过程中，数据类型转换是极为常用的功能。为了更好地支持类型转换，虚拟机提供了一整套专门用于类型转换的指令。这类指令的助记符基本上使用 x2y 的形式给出。其中 x 可能是 i、f、l、d，y 可能是 i、f、l、d、c、s、b。它们的含义如表 11.2 所示。

表 11.2　指令中字符的含义

字　符	含　义	字　符	含　义
i	int	f	float
l	long	d	double
c	char	s	short
b	byte		

比如，i2l 表示将 int 数据转为 long 数据。指令 i2l 在执行时，先将栈顶的 int 数据弹出，然后进行转换，最后，将转化后的 long 型数据压入，如图 11.13 所示，转换后的 long 型数字占用两个字空间。

【示例 11-5】下面这段代码演示一部分这类指令的使用。

```
public void print6(int i) {
    long l=i;
    float f=l;
    int j=(int)l;
}
```

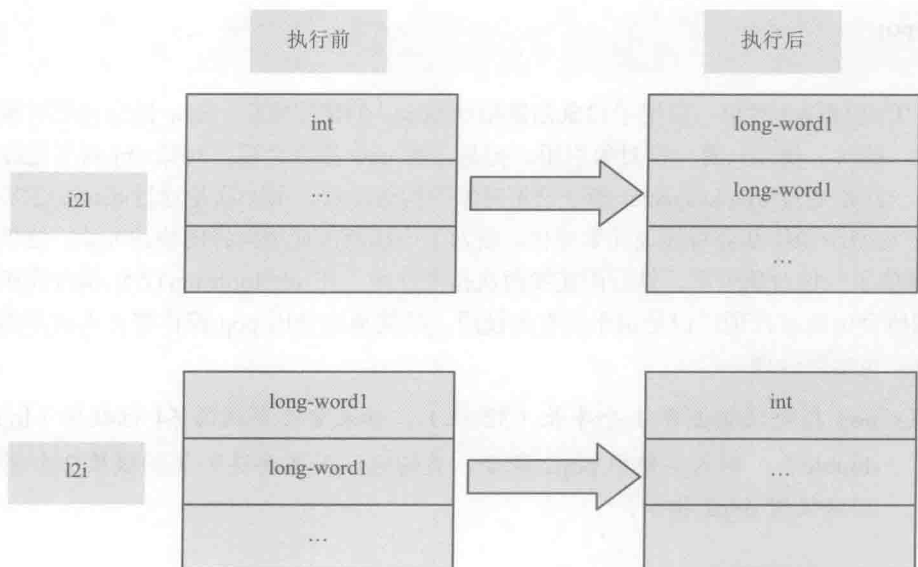


图 11.13　指令 i2l 和 l2i 指令前后操作数栈情况

其字节码指令如下：

```
0: iload_1
1: i2l
2: lstore_2
3: lload_2
4: l2f
5: fstore        4
7: lload_2
8: l2i
9: istore        5
11: return
```

从字节码加粗部分不难看出，int 转换为 long 使用了 i2l，long 转换为 float 使用了 l2f，long 转换为 int 使用了 l2i。

细心的读者也不难发现，在这些转换指令中，只有 i2b、i2c、i2s，但是没有 b2i、c2i 和 s2i。也就说，没有从 byte、char 或是 short 转换为其他数据类型的指令。这不免让人心生疑虑，来看如下代码：

```
public void print7(byte i) {
    int k=i;
```

```
    long l=i;
}
```

这段代码把一个 byte 转为了 int 和 long。如果没有 b2i 和 b2l 这是如何做到的呢？以下是这段代码的字节码表述：

```
0: iload_1        //i 压入栈
1: istore_2       //i 赋值给 k
2: iload_1        //i 入栈
3: i2l            //i 转为 long
4: lstore_3       //long 赋值给 l
5: return
```

我们可以看到，对于 byte 类型转为 int，虚拟机并没有做实质性的转化处理，只是简单地通过操作数栈交换了两个数据。而将 byte 转为 long 时，使用的是 i2l，可以看到在内部 byte 在这里已经等同于 int 处理，类似的还有 short。这种处理方式有两个特点：

- 一方面可以减少实际的数据类型，如果为 short 和 byte 都准备一套指令，那么指令的数量就会大增，而虚拟机目前的设计上，只愿意使用一个字节表示指令，因此指令总数不能超过 256 个，为了节省指令资源，将 short 和 byte 当做 int 处理也在情理之中。
- 另一方面，由于局部变量表中的槽位固定为 32 位，无论是 byte 或者 short 存入局部变量表，都会占用 32 位空间。从这个角度说，也没有必要特意区分这几种数据类型。

11.2.6　运算指令

运算指令为 Java 虚拟机提供了基本的加减乘除等运算功能。基本运行可以分为：加法、减法、乘法、除法、取余、数值取反、位运算、自增运算。每一类指令也有自己支持的数据类型，与前文所述的指令类似，将数据类型使用一个字符表示。

- 加法指令有：iadd、ladd、fadd、dadd。
- 减法指令有：isub、lsub、fsub、dsub。
- 乘法指令有：imul、lmul、fmul、dmul。
- 除法指令有：idiv、ldiv、fdiv、ddiv。
- 取余指令有：irem、lrem、frem、drem。
- 数值取反有：ineg、lneg、fneg、dneg。
- 自增指令：iinc。
- 位运算指令，又可分为如下几种。
    - 位移指令：ishl、ishr、iushr、lshl、lshr、lushr。

- 按位或指令：ior、lor。
- 按位与指令：iand、land。
- 按位异或指令：ixor、lxor。

以乘法指令为例，imul 表示从操作数栈中弹出两个整数，将它们相乘，结果再压入栈。指令 lmul 表示 long 型，fmul 表示对 float 的操作，dmul 表示对 double 的乘法，以此类推。

取余指令用于计算两个数相除后的余数。比如，i%j 就会产生 irem 指令。

【示例 11-6】数值取反指令改变数字的符号位，比如以下 Java 代码，会产生两个 ineg。

```
float i=8;
float j=-i;
i=-j;
```

生成的字节码如下：

```
0: ldc  #73    // float 8.0f
2: fstore_1
3: fload_1
4: fneg
5: fstore_2
6: fload_2
7: fneg
8: fstore_1
9: return
```

指令 fneg 操作前后，操作数栈没有变化，只是栈顶的元素符号位被取反。

指令 iinc 对给定的局部变量做自增操作，这条指令是少数几个执行过程中完全不修改操作数栈的指令。它接收两个操作数：第 1 个为局部变量表的位置，第 2 个为累加数。比如常见的 i++，就会产生这条指令。

【示例 11-7】以下代码对局部变量 i 进行累加 10。

```
public void print11(int j) {
   int i=123;
   i=i+10;
}
```

以上代码产生的字节码如下：

```
0: bipush       123
2: istore_2
```

```
3: iinc        2, 10
6: return
```

由于参数中，this、j 占据了局部变量表的第 0 和第 1 个位置，故 i 处于第 2 个位置。

【示例 11-8】位运算指令由位运算符产生，比如左移<<、无符号右移>>>、右移>>、按位或|、按位与&、异或^、按位取反~等操作产生。不难发现，除了按位取反外，其他位运算都有对应的指令。那么按位取反又是如何工作的呢？以下面的代码为例：

```
int i=123;
int j=~i;
```

生成的字节码指令如下：

```
0: bipush        123
2: istore_1
3: iload_1
4: iconst_m1
5: ixor
6: istore_2
7: return
```

其中，ixor 为整数的按位异或操作，它从栈中弹出两个整数，并将它们按位异或，将结果再压入栈中。可以看到，在 ixor 执行前，压入栈中的数字为 i=123 以及-1（iconst_m1），因此，在虚拟机中，按位取反是通过与-1 异或计算得来的。

注意：-1 的 2 进制表示为一个全 1 的数字 0xFF，任何数字与 0xFF 异或后，自然取反。

11.2.7　对象/数组操作指令

Java 是面向对象的程序设计语言，虚拟机平台从字节码层面就对面向对象做了深层次的支持。有一系列指令专门用于对象操作，可进一步细分为创建指令、字段访问指令、类型检查指令、数组操作指令。

1. 创建指令

创建指令可以用于创建对象或者数组，由于对象和数组在 Java 中的广泛使用，这些指令的使用频率也非常高。创建指令主要有：new、newarray、anewarray 和 multianewarray。

指令 new 用于创建普通对象。它接收一个操作数，为指向常量池的索引，表示要创建的类型，执行完成后，将对象的引用压入栈。

【示例 11-9】指令 newarray 和 anewarray 用来创建数组。前者用于创建基本类型的数组，后者用于创建对象数组。指令 multianewarray 用于创建多维数组。比如以下代码：

```
int[] intarray=new int[10];
Object[] objarray=new Object[10];
int[][] mintarray=new int[10][10];
```

这段代码创建了一个 int 数组，一个 Object 数组和一个 int 二维数组，因此它依次使用了 newarray、anewarray 和 multianewarray。

```
0: bipush        10
2: newarray      int
4: astore_1
5: bipush        10
7: anewarray     #3        // class java/lang/Object
10: astore_2
11: bipush        10
13: bipush        10
15: multianewarray #81,  2 // class "[[I"
19: astore_3
20: return
```

2. 字段访问指令

字段访问指令专门用于访问类或者对象的字段。主要有：getfield、putfield、getstatic、putstatic 4 个。其中，getfield、putfield 用于操作实例对象的字段，getstatic、putstatic 用于操作类的静态字段。以 getstatic 指令为例，它含有一个操作数，为指向常量池的 Fieldref 索引，它的作用就是获取 Fieldref 指定的对象或者值，并将其压入操作数栈。有关 Fieldref 的详细信息，请参阅第 9 章。

以下是常用的控制台输出语句：

```
System.out.println("hello");
```

Java 编译器会为这条语句产生如下 getstatic 指令，用于将 System.out 这个静态字段压入操作数栈。这段指令显示，常量池第 21 号为 Fieldref，它指向 System.out 静态字段，字段类型为 java/io/PrintStream。

```
0: getstatic     #21    // Field java/lang/System.out:Ljava/io/PrintStream;
```

3. 类型检查指令

类型检查指令主要有两个：checkcast 和 instanceof。指令 checkcast 用于检查类型强制转换是否

可以进行。如果可以进行，那么 checkcast 指令不会改变操作数栈，否则它会抛出 ClassCastException 异常。指令 instanceof 用来判断给定对象是否是某一个类的实例，它会将判断结果压入操作数栈。

【示例 11-10】以下代码是 checkcast 和 instanceof 指令的实例。

```
public String chackcast(Object obj) {
    if(obj instanceof String)
        return (String)obj;
    else
        return null;
}
```

上面代码使用了 instanceof 关键字，并使用了强制转换，它们分别会产生 instanceof 和 checkcast 两个字节码。

```
0: aload_1
1: instanceof    #95   // class java/lang/String
4: ifeq          12
7: aload_1
8: checkcast     #95   // class java/lang/String
11: areturn
12: aconst_null
13: areturn
```

可以看到，它们都接收一个操作数，并判断栈顶层元素是否可以转为该操作数给定的类型。

4. 数组操作指令

数组操作指令主要有前文介绍过的 xastore 和 xaload 指令，这里不再赘述。此外，还有一个取数组长度的指令 arraylength，该指令弹出栈顶的数组元素，获取数组的长度，将长度压入栈。

11.2.8 比较控制指令

程序流程离不开条件控制，为了支持条件跳转，虚拟机提供了大量字节码指令，大体上可以分为比较指令、条件跳转指令、比较条件跳转指令、多条件分支跳转、无条件跳转指令等。

1. 比较指令

比较指令的作用是比较栈顶两个元素的大小，并将比较结果入栈。比较指令有：dcmpg、dcmpl、fcmpg、fcmpl、lcmp。与前文讲解的指令类似，首字符 d 表示 double 类型，f 表示 float，l 表示 long。对于 double 和 float 的数字，由于有 NaN 的存在，故各有两个版本，以 float 为例，有 fcmpg 和 fcmpl 两个指令，它们的区别在于在数字比较时，若遇到 NaN 值，处理结果不同。

指令 fcmpg 和 fcmpl 都从栈中弹出两个操作数，并将它们做比较，设栈顶的元素为 v2，栈顶顺位第 2 位的元素为 v1，若 v1=v2，则压入 0，若 v1>v2 则压入 1，若 v1<v2 则压入-1。两个指令的不同之处在于，如果遇到 NaN 值，fcmpg 会压入 1，而 fcmpl 压入-1。

指令 dcmpl 和 dcmpg 也是类似的，根据其命名可以推测其含义，在此不再赘述。指令 lcmp 针对 long 整数，由于 long 整数没有 NaN 值，故无需准备两套指令。

2. 条件跳转指令

条件跳转指令通常和比较指令结合使用。条件跳转指令有：ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull。这些指令都接收两个字节的操作数，用于计算跳转的位置（16 位符号整数作为当前位置的 offset）。它们的统一含义为：弹出栈顶元素，测试它是否满足某一条件，如果满足条件，则跳转到给定位置。在条件跳转指令执行前，一般可以先用比较指令进行栈顶元素的准备，然后再进行条件跳转。

【示例 11-11】比如以下代码：

```
float f1 = 9;
float f2 = 10;
System.out.println(f1 > f2);
```

以上代码产生的指令如下：

```
0:  ldc     #38; //float 9.0f
2:  fstore_1
3:  ldc     #39; //float 10.0f
5:  fstore_2
6:  getstatic #15; //Field java/lang/System.out:Ljava/io/PrintStream;
9:  fload_1
10: fload_2
11: fcmpl
12: ifle    19
15: iconst_1
16: goto    20
19: iconst_0
20: invokevirtual   #30; //Method java/io/PrintStream.println:(Z)V
23: return
```

可以看到，第 9 和第 10 行在栈顶准备了两个比较元素。第 11 行指令对栈顶两个元素进行比较，第 12 行 ifle 获取栈顶的结果，并确认是否需要跳转。

3. 比较条件跳转指令

比较条件跳转指令类似于比较指令和条件跳转指令的结合体，它将比较和跳转两个步骤合二为一，这类指令有：if_icmpeq、if_icmpne、if_icmplt, if_icmpgt、if_icmple、if_icmpge、if_acmpeq 和 if_acmpne。其中指令助记符加上"if_"后，以字符"i"开头的指令针对 int 整数操作（也包括 short 和 byte 类型），以字符"a"开头的指令表示对象引用的比较。

这些指令都接收两个字节的操作数作为参数，用于计算跳转的位置。同时在执行指令时，栈顶需要准备两个元素进行比较。指令执行完成后，栈顶的这两个元素被清空，且没有任何数据入栈。如果预设条件成立，则执行跳转，否则，继续执行下一条语句。

【示例 11-12】以下面的代码为例，比较 short 和 byte 两个数字的大小。

```
short f1 = 9;
byte f2 = 10;
System.out.println(f1 > f2);
```

生成的字节码如下：

```
0:   bipush  9
2:   istore_1
3:   bipush  10
5:   istore_2
6:   getstatic    #15; //Field java/lang/System.out:Ljava/io/PrintStream;
9:   iload_1
10:  iload_2
11:  if_icmple 18
14:  iconst_1
15:  goto   19
18:  iconst_0
19:  invokevirtual   #30; //Method java/io/PrintStream.println:(Z)V
22:  return
```

其中，第 9 和第 10 行将需要比较的数字压入栈，第 11 行执行比较，如果条件成立则跳转 18 行输出 0，否则继续执行后一条指令输入 1。

【示例 11-13】如果比较的元素是对象，那么就会使用 if_acmpeq 和 if_acmpne 指令。

```
Object f1 = new Object();
Object f2 = new Object();
System.out.println(f1 == f2);
System.out.println(f1 != f2);
```

以上代码生成的部分指令为（篇幅所限没有给出完整指令）：

```
19:  aload_1
20:  aload_2
21:  if_acmpne      28
24:  iconst_1
25:  goto    29
………
35:  aload_1
36:  aload_2
37:  if_acmpeq      44
40:  iconst_1
41:  goto    45
44:  iconst_0
45:  invokevirtual   #30; //Method java/io/PrintStream.println:(Z)V
48:  return
```

其中第 9、10 行和第 35、36 行，分别压入栈顶元素进行比较，第 21 和第 37 行执行对象引用的比较并确认是否需要跳转。

4. 多条件分支跳转

多条件分支跳转指令是专为 switch-case 语句设计的，主要有 tableswitch 和 lookupswitch。从助记符上看，两者都是 switch 语句的实现，它们的区别在于 tableswitch 要求多个条件分支值是连续的，它内部只存放起始值和终止值，以及若干个跳转偏移量，通过给定的操作数 index，可以立即定位到跳转偏移量上，因此效率比较高。指令 lookupswitch 内部存放着各个离散的 case-offset 对，每次执行都要搜索全部的 case-offset 对，找到匹配的 case 值，并根据对应的 offset 计算跳转地址，因此效率较低。

指令 tableswitch 的示意图如图 11.14 所示。由于 tableswitch 的 case 值是连续的，因此只需要记录最低值和最高值，以及每一项对应的 offset 偏移量，给定的 index 值通过简单的计算即可直接定位到 offset。

指令 lookupswitch 处理的是离散的 case 值，但是出于效率考虑，将 case-offset 对按照 case 值大小排序，给定 index 时，需要查找与 index 相等的 case，获得其 offset，如果找不到则跳转到 default。指令 lookupswitch 如图 11.15 所示。

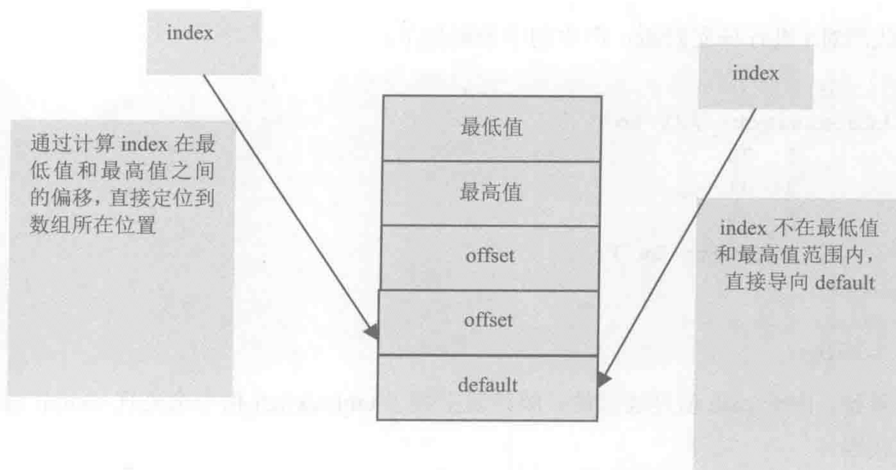


图 11.14　tableswitch 示意图

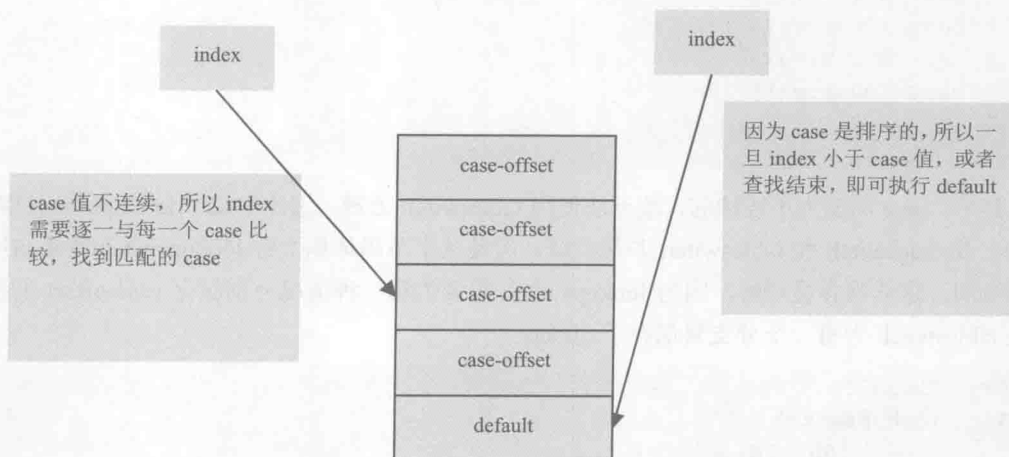


图 11.15　lookupswitch 示意图

【示例 11-14】下面以几个代码片段为例，介绍这两个指令的具体使用方式。

```
public void swtich1(int i) {
    switch (i) {
    case 1: break;
    case 2: break;
    case 3: break;
    }
}
```

上述代码对 i 进行分支判断，产生的字节码如下：

```
0:   iload_1
1:   tableswitch{ //1 to 3
             1: 28;
             2: 31;
             3: 34;
             default: 34 }
28:  goto    34
31:  goto    34
34:  return
```

可以看到，由于 case 值是连续的，编译器生成了 tableswitch 指令来处理 switch 语句。再来看第 2 段代码：

```
public void swtich2(int i) {
    switch (i) {
    case 100:break;
    case 200:break;
    case 300:break;
    }
}
```

其中，case 的值为不连续的，故无法使用 tableswitch 处理，因此生成了 lookupswitch 指令。形式上 lookupswitch 和 tableswitch 非常类似，但是从字节码体积上看 lookupswitch 需要占用更多的空间。这非常容易理解，因为 lookupswitch 需要为每一种情况分别保存 case-offset 的值，但是 tableswitch 为每一个分支只保存了 offset。

```
0:   iload_1
1:   lookupswitch{ //3
             100: 36;
             200: 39;
             300: 42;
             default: 42 }
36:  goto    42
39:  goto    42
42:  return
```

【示例 11-15】在 JDK 1.7 后，swtich 语句得到了增强，目前已经支持使用字符串进行 switch 操作。而对字符串的 switch-case 语句是通过 lookupswitch 指令来实现的。以下面代码为例：

```
public void swtich4(String i) {
    switch (i) {
```

```
        case "geym":
            break;
        case "zbase":
            break;
        case "java":
            break;
        default:
        }
    }
```

上述代码用字符串作为 switch 对象，它所生成的字节码如下：

```
0:  aload_1
1:  dup
2:  astore_2
3:  invokevirtual   #51; //Method java/lang/String.hashCode:()I
6:  lookupswitch{ //3
        3169394: 40;
        3254818: 52;
        115685963: 64;
        default: 73 }
40: aload_2
41: ldc     #57; //String geym
43: invokevirtual   #59; //Method java/lang/String.equals:(Ljava/lang/Object;)Z
46: ifne    73
49: goto    73
52: aload_2
53: ldc     #63; //String java
55: invokevirtual   #59; //Method java/lang/String.equals:(Ljava/lang/Object;)Z
58: ifne    73
61: goto    73
64: aload_2
65: ldc     #65; //String zbase
67: invokevirtual   #59; //Method java/lang/String.equals:(Ljava/lang/Object;)Z
70: ifne    73
73: return
```

可以看到，为了支持对 String 的 switch 操作，在字节码第 3 行（实为偏移量，用行简化描述）调用了字符串的 hashCode()方法，得到 int 整数。在 lookupswitch 指令中，实际使用该 hash 值作为分支的 case。如果 hash 值没有匹配的，则必然字符串也没有匹配，因此可以直接执行 default 处的指令，但如果 hash 值匹配，考虑到 hash 冲突的存在，这里并没有立即进行匹配后

的指令，还是对匹配的项进行二次确认，在第 43 行使用 String.equals()函数判断字符串是否真的相等。如果确实相等，则执行对应的语句，否则跳转退出。由此可见，当使用 String 作为 case 类型时，虚拟机要多执行 hash 计算以及字符串相等等操作，性能必然要低于直接对 int 的处理。

5. 无条件跳转

目前主要的无条件跳转指令为 goto。指令 jsr、ret 虽然也是无条件跳转的，但主要用于 try-finally 语句，且已经被虚拟机逐渐废弃，故不在本书中介绍这两个指令。指令 goto 接收两个字节的操作数，共同组成一个带符号的整数，用于指定指令的偏移量，指令执行的目的就是跳转到偏移量给定的位置处。如果指令偏移量太大，超过双字节的带符号整数的范围，则可以使用指令 goto_w，它和 goto 有相同的作用，但是它接收 4 个字节作为操作数，可以表达更宽的地址范围。

11.2.9 函数调用与返回指令

为了支持函数调用和返回值的处理，虚拟机提供了一系列基本指令。就函数调用而言，有指令 invokevirtual、invokeinterface、invokespecial、invokestatic 和 invokedynamic 几种。在函数返回时，需要将返回值压入调用者操作数栈，此时需要使用 xreturn 指令（x 可以是 i 、l、f、d、a 或空）。

这些 invokeXXX 指令都是进行函数调用的，但是各自均有自己的使用范围。

- invokevirtual：虚函数调用，调用对象的实例方法，根据对象的实际类型进行派发，支持多态，也是最常见的 Java 函数调用方式。
- invokeinterface：指接口方法的调用，当被调用对象申明为接口时，使用该指令调用接口的方法。
- invokespecial：调用特殊的一些方法，比如构造函数、类的私有方法、父类方法。这些方法都是静态类型绑定的，不会在调用时进行动态派发。
- invokestatic：调用类的静态方法，这个也是静态绑定的。
- invokedynamic：调用动态绑定的方法，这个是 JDK 1.7 后新加入的指令，该指令的具体介绍请参考本章 11.5 节。

函数调用结束前，需要进行返回。返回时，需要使用 xreturn 指令将返回值存入调用者的操作数栈中。根据返回值类型的不同，该指令的前缀有所不同，当返回 int 时，使用指令 ireturn，当返回值为 void 时，使用指令 return。该指令被调用时，如果方法是同步的，那么调用后，监视器锁将被释放。

【示例 11-16】下面通过几个代码片段，来理解一下这几个指令的使用方式。

首先，来介绍一下指令 invokevirtual 的使用。

```
System.out.println("aa");
```

以上语句会产生：

```
0:   getstatic      #16; //Field java/lang/System.out:Ljava/io/PrintStream;
3:   ldc    #22; //String aa
5:   invokevirtual  #24; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

注意指令 invokevirtual，在这里它调用了 PrintStream 实例的 println()方法。在调用前，操作数栈中将压入调用对象的实例，以及该函数的所有参数。在本例中为 System.out 实例和字符“aa”。指令 invokevirtual 需要两个字节作为操作数，用于计算指向常量池的索引，这里索引必须指向 CONSTANT_Methodref 入口，表示需要调用的方法。

指令 invokeinterface 指调用接口的函数。比如：

```
Thread t=new Thread();
t.run();
((Runnable)t).run();
```

上述代码生成了 Thread 对象，并调用了它的 run()方法。Thread 类实现了 Runnable 接口。这里使用两种方式调用 run()：第 1 种直接在 Thread 申明的实例上调用；第 2 种将其转为接口类型 Runnable，再进行调用。这两种调用方式使用的 invoke 指令是不同的，如下指令代码：

```
0:   new    #45;                 //class java/lang/Thread
3:   dup
4:   invokespecial   #47;        //Method java/lang/Thread."<init>":()V
7:   astore_1
8:   aload_1
9:   invokevirtual   #48;        //Method java/lang/Thread.run:()V
12:  aload_1
13:  invokeinterface #51,  1;    //InterfaceMethod java/lang/Runnable.run:()V
18:  return
```

字节码指令中第 9 行，使用 invokevirtual 指令，是直接针对 Thread 对象的调用，第 13 行，则是针对 Runnable 接口的调用。和 invokevirtual 不同，invokeinterface 在调用时，需要额外传入 1 个字节，作为无符号整数，表示这次函数调用所需参数的字数（1 字为 32 位），包含隐含的 this。在本例中，函数没有参数，只需要当前引用 this，故字数为 1。

指令 invokespecial 用于调用特殊的函数。比如调用类的构造函数、私有方法或者父类方法

时都会使用该指令。该指令调用时，接收两个字节作为其操作数，用于计算常量池索引入口，且该入口必须为 CONSTANT_Methodref。此外，在调用时，必须在操作数栈中准备好对象实例、函数参数等信息。

一次简单的对象实例创建，就需要调用构造函数，比如：

```
Date d=new Date();
```

以上语句会产生：

```
0:   new     #31; //class java/util/Date
3:   dup
4:   invokespecial   #33; //Method java/util/Date."<init>":()V
7:   astore_1
```

可以看到，使用指令 invokespecial 调用了 Date 类的构造函数。

【示例 11-17】下面，再来看一个调用类的私有方法的例子。由于类的私有方法不具有多态性，即使在子类中有相同签名的私有方法，也不能覆盖父类中对应的私有方法的行为，因此对于私有方法调用可以使用静态绑定。

```
private void pMethod(){

}
public void invokeSpecial2(){
   pMethod();
}
```

以上代码调用 pMethod()私有方法，它产生的字节码如下：

```
0:   aload_0
1:   invokespecial   #37; //Method pMethod:()V
4:   return
```

此外，调用父类方法时也使用 invokespecial，比如：

```
super.toString();
```

会生成：

```
0:   aload_0
1:   invokespecial   #40; //Method java/lang/Object.toString:()Ljava/lang/
String;
```

可以看到，invokespecial 在调用父类方式，通过操作数直接指向父类的 toString()方法，从

而避免了子类 toString()方法的使用。

11.2.10　同步控制

为实现多线程的同步，Java 虚拟机还提供了 monitorenter、monitorexit 两条执行来完成临界区的进入和离开操作。当一个线程进入同步块时，它使用 monitorenter 指令请求进入，如果当前对象的监视器计数器为 0，则它会被准许进入，若为 1，则判断持有当前监视器的线程是否为自己，如果是，则进入，否则进行等待，直到对象的监视器计数器为 0，才会被允许进入同步块。当线程退出同步块时，需要使用 monitorexit 申明退出。在 Java 虚拟机中，任何对象都有一个监视器与之相关联，用来判断对象是否被锁定，当监视器被持有后，对象处于锁定状态。

指令 monitorenter 和 monitorexit 在执行时，都需要在操作数栈顶压入对象，之后，monitorenter 和 monitorexit 的锁定和释放都是针对这个对象的监视器进行的。

图 11.16 展示了监视器如何保护临界区代码不同时被多个线程访问，只有当线程 4 离开临界区后，线程 1、2、3 才有可能进入。

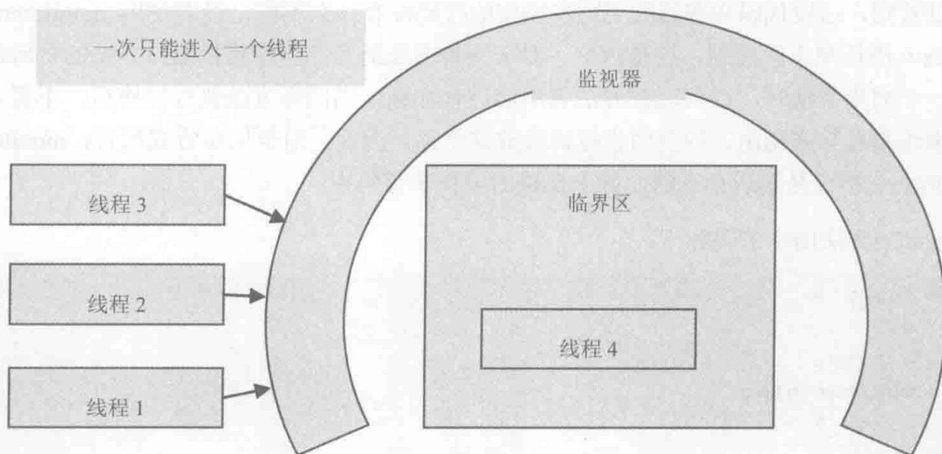


图 11.16　监视器保护临界区

【示例 11-18】这里给出一个 monitorenter、monitorexit 指令的使用示例。

```
public class SyncAdd {
    private int i=0;
    public synchronized void add1(){
        i++;
    }
```

```
    public void add2(){
        synchronized(this){
            i++;
        }
    }
}
```

在类 SyncAdd 方法中，有两个方法 add1()和 add2()，它们都对当前 this 对象进行加锁，并对实例字段 i 进行更新。对于 add1()方法，有如下字节码：

```
0:  aload_0
1:  dup
2:  getfield        #12; //Field i:I
5:  iconst_1
6:  iadd
7:  putfield        #12; //Field i:I
10: return
```

可以看到，这段代码和普通的无同步操作的代码没有什么不同，没有使用 monitorenter 和 monitorexit 进行同步区控制，这是因为，对于同步方法而言，当虚拟机通过方法的访问标示符判断是一个同步方法时，会自动在方法调用前进行加锁，当同步方法执行完毕后，不管方法是正常结束还是有异常抛出，均会由虚拟机释放这个锁。因此，对于同步方式而言，monitorenter 和 monitorexit 指令是隐式存在的，并未直接出现在字节码中。

而 add2()方法的字节码如下：

```
0:  aload_0
1:  dup
2:  astore_1
3:  monitorenter
4:  aload_0
5:  dup
6:  getfield        #12; //Field i:I
9:  iconst_1
10: iadd
11: putfield        #12; //Field i:I
14: aload_1
15: monitorexit
16: goto    22
19: aload_1
20: monitorexit
```

```
    21:  athrow
    22:  return
   Exception table:
    from    to  target type
      4    16    19   any
     19    21    19   any
```

可以看到，对于同步块而言，字节码中已经生成了 monitorenter 和 monitorexit 指令。读者也许会觉得奇怪，为什么一个 monitorenter 居然会对应两个 monitorexit 指令。读者可以注意字节码最后给出的异常表，虽然在源码中没有进行异常处理，但是在生成的字节码中自动插入了一段异常处理代码，异常处理点为第 19 行。整段字节码的解析如下：

第 0 行将 this 引用入栈。

第 1 行复制 this 引用，并入栈。

第 2 行将 this 引用弹出，存入第 1 个局部变量。

第 3 行根据栈顶的 this 引用进行加锁。

第 4~14 行执行了 i++操作。

第 15 行表示释放锁，此时，i++已经完成。

第 16 行跳转到第 22 行，并退出。

如果在第 4~16 行执行期间，遇到任何异常，则进入第 19 行处理。

第 19 行将第 1 个局部变量入栈，该变量就是 this，由第 2 行存入。

第 20 行根据栈顶的 this，退出临界区，释放锁。

第 21 行抛出当前发生的异常，异常对象位于栈顶。

11.2.11　再看 Class 的方法结构

在 9.2.14 节中，以 SimpleUser.setId()方法为例，分析了 Class 文件的静态结构。读者可以回顾相关内容。图 11.17 为 setId()方法的结构和指令部分。

```
00000300  00 00 0C 00 01 00 00 00  05 00 13 00 14 00 00 00
00000310  01 00 19 00 1A 00 02 00  1B 00 00 00 04 00 01 00
00000320  1C 00 0E 00 00 00 73 00  02 00 03 00 00 00 14 2A
00000330  1B B5 00 17 A7 00 0E 4D  B2 00 1E 2C B6 00 24 B6
00000340  00 28 B1 00 01 00 00 00  05 00 08 00 1C 00 03 00
00000350  11 00 00 00 12 00 04 00  00 00 0F 00 05 00 10 00
00000360  09 00 11 00 13 00 13 00  12 00 00 00 20 00 03 00
```

图 11.17　setId()方法部分字节码分析

其中细线框部分为方法的字节码指令。为阅读方便，这里再次给出 SimpleUser.setId()的方法源码。

```
public void setId(int id) throws IllegalStateException {
    try {
        this.id = id;
    } catch (IllegalStateException e) {
        System.out.println(e.toString());
    }
}
```

解析这段字节码指令，如表 11.3 所示。

表 11.3　setId()方法字节码解析

字　节	指　令	含　义
2A	aload_0	局部变量表第0位压栈 this
1B	aload_1	局部变量表第1位压栈 参数 int
B5	putfield	23号常量池为Field ，名称为id ，类型为int
0017	putfield的操作数	
A7	goto	跳转到偏移量19处
000E	goto的操作数	
4D	astore_2	栈顶元素存入局部变量2，即给异常e赋值
B2	getstatic	30号常量池为java/lang/System.out:Ljava/io/PrintStream
001E	getstatic操作数	
2C	aload_2	局部变量2压栈，即异常e压栈
B6	invokevirtual	36号常量池为IllegalStateException.toString()方法，这里调用这个toString()方法
0024	invokevirtual操作数	
B6	invokevirtual	40号常量池为PrintStream.println()方法，这里调用println()进行输出
0028	invokevirtual操作数	
B1	return	方法返回

注意：表中第 1 列的数字均为 16 进制，故在常量池索引计算时，务必先转为 10 进制。

最后，给出这段代码片段的完整字节码信息，方便读者查看比对。

```
0:   aload_0
1:   iload_1
2:   putfield      #23; //Field id:I
5:   goto    19
8:   astore_2
9:   getstatic     #30; //Field java/lang/System.out:Ljava/io/PrintStream;
12:  aload_2
```

```
   13:  invokevirtual   #36; //Method java/lang/IllegalStateException.toString:
()Ljava/lang/String;
   16:  invokevirtual   #40; //Method java/io/PrintStream.println:(Ljava/lang/
String;)V
   19:  return
     Exception table:
      from   to  target type
        0     5     8   Class java/lang/IllegalStateException
```

11.3　更上一层楼：再看 ASM

读到这里，读者或许会觉得学习虚拟机的指令枯燥而乏味，了解 Class 文件错综复杂的结构似乎也没有太大的价值。而事实恰恰相反，这些看似枯燥的知识，可以让我们做很多有趣的事情。比如大名鼎鼎的 CGLIB，其底层就是基于 ASM 的，而要用好 ASM，了解 Class 文件结构和常用指令是必不可少的。通过 ASM 的字节码操作，可以动态创建新的类型，可以为类增加新的功能。虽然使用 CGLIB 这些高级库也可以完成大量的工作，但是直接使用 ASM 还是大有好处：ASM 的性能最好，灵活度最高，功能也最为强大，可以将操作粒度定位到每一条指令。

11.3.1　为类增加安全控制

【示例 11-19】本节将介绍一个使用 ASM 的基本案例。现有一个账户类，可以进行某些操作如下：

```
public class Account {
    public void operation() {
        System.out.println("operation...");
    }
}
```

目前，这个 operation()方法没有任何控制手段，现在希望为 operation()方法增加一些安全校验，以判断这个对象是否有权限执行这个方法，如果有，则执行该方法，如果没有，则直接退出。当然，这一切是要在不修改 Account 类源码的前提下进行的。

注意： 根据开闭原则，一个系统对功能的增加应该是开放的，对修改应该是闭合的。也就是说系统应该拥有一定的扩展性，但是也应该尽可能不要修改原有系统的代码。对于线上系统，修改代码带来的最大问题是很容易将原有系统破坏，需要安排更多的回归测试时间。从另一方面讲，如果系统中要进行权限校验的功

能点很多，那么如果能进行统一的批量处理，要比逐个处理方便、高效、精确得多。

假设现在要增加的权限校验函数为：

```
public class SecurityChecker {
    public static boolean checkSecurity() {
        System.out.println("SecurityChecker.checkSecurity ...");
        if((System.currentTimeMillis()&0x1)==0)
            return false;
        else
            return true;
    }
}
```

以上代码模拟了权限校验，通过随机方式给出是否校验通过。现在系统要做的就是将SecurityChecker.checkSecurity()函数置于 Account.operation()函数之前运行，如果权限校验失败，则阻止 Account.operation()继续处理。

下面的 SecurityWeaveGenerator 类，将 checkSecurity()函数放置到 operation()函数的第 1 行执行。它的核心是代码第 6 行的 AddSecurityCheckClassAdapter 类。这是一个 ClassVisitor，它负责实际的字节码织入操作。在代码的第 8~12 行，将新生成的 Account 类写入文件，覆盖由 Java 编译器产生的 Account 类的 Class 文件。

```
01 public class SecurityWeaveGenerator{
02   public static void main(String args[]) throws Exception {
03       String className=Account.class.getName();
04       ClassReader cr = new ClassReader(className);
05       ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS|ClassWriter.
COMPUTE_FRAMES);
06       AddSecurityCheckClassAdapter classAdapter = new
AddSecurityCheckClassAdapter(cw);
07       cr.accept(classAdapter, ClassReader.SKIP_DEBUG);
08       byte[] data = cw.toByteArray();
09       File file = new File("bin/"+className.replaceAll("\\.", "/")+".class");
10       FileOutputStream fout = new FileOutputStream(file);
11       fout.write(data);
12       fout.close();
13   }
14 }
```

AddSecurityCheckClassAdapter 类的内容如下代码所示：

```
01 class AddSecurityCheckClassAdapter extends ClassVisitor {
02     public AddSecurityCheckClassAdapter( ClassVisitor cv) {
03         super(Opcodes.ASM5, cv);
04     }
05
06     public MethodVisitor visitMethod(final int access, final String name,
07         final String desc, final String signature, final String[] exceptions) {
08         MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
exceptions);
09         MethodVisitor wrappedMv = mv;
10         if (mv != null) {
11             if (name.equals("operation")) {
12                 wrappedMv = new AddSecurityCheckMethodAdapter(mv);
13             }
14         }
15         return wrappedMv;
16     }
17 }
```

在 AddSecurityCheckClassAdapter 中，覆盖了 visitMethod()方法，在代码第 11 行，当访问到 operation 方法时，交由 AddSecurityCheckMethodAdapter 类处理。AddSecurityCheckMethodAdapter 是一个 MethodVisitor，它负责最终的字节码修改。

```
class AddSecurityCheckMethodAdapter extends MethodVisitor {
    public AddSecurityCheckMethodAdapter(MethodVisitor mv) {
        super(Opcodes.ASM5,mv);
    }

    public void visitCode() {
        Label continueLabel = new Label();
        visitMethodInsn(Opcodes.INVOKESTATIC, "geym/zbase/ch11/aop/securitycheck/SecurityChecker",
            "checkSecurity", "()Z");
        visitJumpInsn(Opcodes.IFNE,continueLabel);
        visitInsn(Opcodes.RETURN);
        visitLabel(continueLabel);
        super.visitCode();
    }
}
```

AddSecurityCheckMethodAdapter 类覆盖了 MethodVisitor 的 visitCode()方法。当访问到方法的字节码时，在第 7~12 行，织入了对 SecurityChecker. checkSecurity()的调用。如果 checkSecurity()返回了 false（栈顶为 0），根据指令 ifne，跳转不会发生，程序返回。如果 checkSecurity()返回了 true（栈顶为 1），则指令 ifne 发生跳转，继续执行原先的 operation()操作。

首先使用 javac 编译并生成 Account.class，接着使用 SecurityWeaveGenerator 类对 Account.class 进行处理，织入权限校验的字节码，最后使用以下简单的代码测试：

```
public class RunAccountMain {
    public static void main(String[] args) {
        Account account = new Account();
        account.operation();
    }
}
```

程序可能输出如下：

```
SecurityChecker.checkSecurity ...
operation...
```

11.3.2 统计函数执行时间

当系统遇到性能问题时，获取函数的执行时间有助于帮助排查性能问题。统计函数执行时间最简单的方式是在函数入口和出口处都使用 System.currentTimeMillis()函数获得系统时间，然后计算两者的差值。但一般情况下，不可能为每一个函数都写这样的语句，实际工作中，往往不会有这样的信息统计。但得益于 ASM 框架，现在，可以在不修改源码、不改变原有系统的情况下，直接将统计函数织入系统。

【示例 11-20】下面代码使用 Thread.sleep()模拟一段耗时的函数调用，现在 Acccout.operation()本身并不具有计时功能。

```
public class Account {
    public void operation() {
        System.out.println("operation....");
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

现在，需要加入的计时统计功能如下：

```
public class TimeStat {
    static ThreadLocal<Long> t=new ThreadLocal<Long>();
    public static void start() {
        t.set(System.currentTimeMillis());
    }

    public static void end(){
        long time=System.currentTimeMillis()-t.get();
        System.out.print(Thread.currentThread().getStackTrace()[2]+" spend:");
        System.out.println(time);
    }
}
```

类 TimeStat 实现了函数的调用计时，在进入函数时，可以使用 start()方法表示函数调用开始，在离开函数时，使用 end()方法表示函数调用结束。函数调用结束后，打印出当前正在调用的函数名称以及实际的系统耗时。

以下代码将 TimeStat.start()和 TimeStat.end()织入 Account. operation()中。

```
01 public class TimeStatWeaveGenerator{
02     public static void main(String args[]) throws Exception {
03         String className=Account.class.getName();
04         ClassReader cr = new ClassReader(className);
05         ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS|ClassWriter.COMPUTE_FRAMES);
06         TimeStatClassAdapter classAdapter = new TimeStatClassAdapter(cw);
07         cr.accept(classAdapter, ClassReader.SKIP_DEBUG);
08         byte[] data = cw.toByteArray();
09         File file = new File("bin/"+className.replaceAll("\\.", "/")+".class");
10         FileOutputStream fout = new FileOutputStream(file);
11         fout.write(data);
12         fout.close();
13     }
14 }
```

以上代码中，第 6 行使用 TimeStatClassAdapter 类，完成具体的字节码修改工作。修改完成后，在第 9~12 行写入 Class 文件，覆盖原有的 Class 文件。

TimeStatClassAdapter 是一个 ClassVisitor，这里，需要覆盖它的 visitMethod()方法，对 operation()方法进行修改。

```
class TimeStatClassAdapter extends ClassVisitor {
    public TimeStatClassAdapter( ClassVisitor cv) {
        super(Opcodes.ASM5, cv);
    }
    public MethodVisitor visitMethod(final int access, final String name,
        final String desc, final String signature, final String[] exceptions) {
        MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
exceptions);
        MethodVisitor wrappedMv = mv;
        if (mv != null) {
            if (name.equals("operation")) {
                wrappedMv = new TimeStatMethodAdapter(mv);
            }
        }
        return wrappedMv;
    }
}
```

以上代码在访问方法时，判断是否是 operation()方法，如果是，则进行方法字节码的调整，并将这个工作委托给 TimeStatMethodAdapter 完成。下面是 TimeStatMethodAdapter 的实现。

```
class TimeStatMethodAdapter extends MethodVisitor implements Opcodes {
    public TimeStatMethodAdapter(MethodVisitor mv) {
        super(Opcodes.ASM5, mv);
    }

    public void visitCode() {
            visitMethodInsn(Opcodes.INVOKESTATIC, "geym/zbase/ch11/aop/timestat/TimeStat", "start", "()V");
        super.visitCode();
    }

    @Override
    public void visitInsn(int opcode) {
        if ((opcode >= IRETURN && opcode <= RETURN)) {
            visitMethodInsn(Opcodes.INVOKESTATIC, "geym/zbase/ch11/aop/timestat/TimeStat", "end", "()V");
        }
        mv.visitInsn(opcode);
    }
}
```

其中，第 6 行的 visitCode()在方法的 Code 属性被访问时调用，因此，在这里插入对 TimeStat.start()方法的调用，表示方法的开始。在第 12 行，覆盖了 visitInsn()函数，当访问到 xreturn 指令时，进行 TimeStat.end()函数调用，表示方法即将退出。从字节码的指令上看多个 xreturn 指令是连续的，如下所示：

```
ireturn= 172, // 0xac
lreturn= 173, // 0xad
freturn= 174, // 0xae
dreturn= 175, // 0xaf
areturn= 176, // 0xb0
return = 177, // 0xb1
```

因此在 visitInsn()中，简单地通过指令值获得范围，判断是否为 xreturn 函数返回指令。使用 TimeStatWeaveGenerator 修改 Account.class，将时间统计的字节码进行织入，并运行以下代码：

```
public class RunTimeStatMainAfterGen {
    public static void main(String[] args) {
        Account account = new Account();
        account.operation();
    }
}
```

可以得到输出如下：

```
operation....
geym.zbase.ch11.aop.timestat.Account.operation(Unknown Source) spend:10
```

可以看到在 operation()函数调用结束后，程序还输出了函数运行的耗时。

11.4 谁说 Java 太刻板：Java Agent 运行时修改类

在 JDK 1.5 时，引入了 java.lang.Instrument 包，该包提供了一些工具帮助开发人员在 java 程序运行时，动态修改系统中的 Class 类型。其中，使用该软件包的一个关键组件为 Java Agent。从名字看，似乎可以理解成 Java 代理，而实际上，它的功能更像是 Class 类型的转换器，它可以在运行时接收程序外部请求，对 Class 类型进行修改。

在命令行执行 java，可以看到 java 命令的帮助，其中，不难发现有一个 javaagent 的选项。

```
-javaagent:<jarpath>[=<options>]
              加载 Java 编程语言代理，请参阅 java.lang.instrument
```

也就是说，Java 程序在运行时，可以指定一个 Java Agent 作为其编程语言代理。

11.4.1 使用-javaagent 参数启动 Java 虚拟机

参数-javaagent 可以用于指定一个 jar 包，并且对该 jar 包有如下要求：

（1）这个 jar 包的 MANIFEST.MF 文件必须指定 Premain-Class 项。

（2）Premain-Class 指定的那个类必须实现 premain()方法。

Premain-Class 仅从英文单词字面上理解，就是运行在 main()函数之前的类。当 Java 虚拟机启动时，在执行 main()函数之前，虚拟机会先运行-javaagent 所指定 jar 包内 Premain-Class 这个类的 premain()方法。其中，premain()方法的签名如下：

```
public static void premain(String agentArgs, Instrumentation inst)
```

其中，agentArgs 是通过命令行传给 Java Agent 的参数，inst 提供 Java Class 字节码转换的工具。Instrumentation 类的常用 API 如下：

- void addTransformer(ClassFileTransformer transformer, boolean canRetransform)
  增加一个 Class 文件的转换器，转换器用于改变 Class 二进制流的数据，参数 canRetransform 设置是否允许重新转换。
- void redefineClasses(ClassDefinition... definitions)
  在类加载之前，重新定义 Class 文件。ClassDefinition 表示对一个类新的定义。如果在类加载之后，需要使用 retransformClasses()函数重新定义类。
- boolean removeTransformer(ClassFileTransformer transformer)
  移出一个类转换器。
- void retransformClasses(Class<?>... classes)
  在类加载之后，重新定义 Class。

【示例 11-21】下面的例子定义了一个 Java Agent，这个 Agent 打印出在 main()函数运行后，系统载入的类型。

```
public class PreMainTraceAgent {
  public static void premain(String agentArgs, Instrumentation inst)
          throws ClassNotFoundException, UnmodifiableClassException {
    System.out.println("agentArgs:"+agentArgs);
      inst.addTransformer(new ClassFileTransformer(){
       @Override
        public byte[] transform(ClassLoader loader, String className,
```

```
Class<?> classBeingRedefined,
   08                ProtectionDomain protectionDomain, byte[] classfileBuffer)
throws IllegalClassFormatException {
   09             System.out.println("load Class:"+className);
   10             return classfileBuffer;
   11         }
   12       });
   13    }
   14 }
```

上述代码第 2 行，定义了 Agent 的 premain()方法，该方法会在 main()函数执行前调用。

代码第 4 行，打印了传递给 Agent 的参数。

代码第 5 行，加入了一个类转换器。这里使用匿名内部类，定义了一个类转换器。这个类转换器很简单，只是简单地打印出获取的类名，不做实质性转换。

代码第 7 行为类转换器接口的 transform()方法，用于完成类的转换。该接口函数定义如下：

```
byte[] transform(ClassLoader         loader,
                 String              className,
                 Class<?>            classBeingRedefined,
                 ProtectionDomain    protectionDomain,
                 byte[]   classfileBuffer) throws IllegalClassFormatException;
```

其中，loader 为定义类的类加载器，className 表示类的全限定名，比如“java/lang/String”。参数 classBeingRedefined 表示：如果是被重定义或重转换触发，则为重定义或重转换的类；如果是类加载，则为 null。参数 protectionDomain 表示要定义或重定义的类的保护域。参数 classfileBuffer 表示类文件格式的二进制数据（只读，不能修改）。该函数的返回值为重新定义的新的类的二进制数据。

将以上的 Java Agent 打包成 jat.jar。并设置 META-INF/ MANIFEST.MF 文件如下：

```
Manifest-Version: 1.0
Premain-Class: geym.zbase.ch11.agent.PreMainTraceAgent
Can-Redine-Classes: true
Can-Retransform-Classes: true
```

这就完成了一个保存有合法 Java Agent 的 jar 包。使用以下代码测试这个 jat.jar：

```
public class RunAccountMain {
    public static void main(String[] args) {
        Account account = new Account();
```

```
            account.operation();
        }
    }
```

上述代码中的 Account，为前文 11.3.2 节中的 Account 类。使用 Java 虚拟机参数：-javaagent:d:/jat.jar 运行代码，输出如下：

```
agentArgs:null
load Class:sun/launcher/LauncherHelper
load Class:geym/zbase/ch11/agent/RunAccountMain
load Class:java/lang/Void
load Class:geym/zbase/ch11/aop/timestat/Account
load Class:java/lang/InterruptedException
operation....
load Class:java/lang/Shutdown
load Class:java/lang/Shutdown$Lock
```

可以看到，PreMainTraceAgent 已经可以正常工作了，这里所加载的类型，并非系统中所有的 Class 类。实际上，在 jat.jar 执行之前的所有已加载的类型都是无法捕获的。因此，大量的系统核心类会丢失，但是应用程序的类都在 main()函数执行之后加载，因此都可以通过这种方式捕获，如上述代码的字体加粗部分。

11.4.2 使用 Java Agent 为函数增加计时功能

在前文 11.3.2 节中，已经通过 ASM 进行字节码织入，使 Account.operation()方法具有计时功能。但提供的织入方式不够灵活，要求在 Account 类在进行 javac 编译后，再进行基于文件的织入。这种做法虽然可行，但是过于繁琐。而有了 Java Agent 的支持，就可以大大改进这种方式。

【示例 11-22】下面的代码展示了通过 Java Agent 直接进行字节码织入的方式。

```
01 public class PreMainAddTimeStatAgent {
02     public static void premain(String agentArgs, Instrumentation inst)
03             throws ClassNotFoundException, UnmodifiableClassException {
04       System.out.println("agentArgs:"+agentArgs);
05         inst.addTransformer(new ClassFileTransformer(){
06          @Override
07            public byte[] transform(ClassLoader loader, String className,
Class<?> classBeingRedefined,
08                   ProtectionDomain protectionDomain, byte[] classfileBuffer)
throws IllegalClassFormatException {
```

```
09            if(className.equals("geym/zbase/ch11/aop/timestat/Account")){
10                System.out.println("meet geym.zbase.ch11.aop.timestat. Account");
11                ClassReader cr = new ClassReader(classfileBuffer);
12                ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS|
ClassWriter.COMPUTE_FRAMES);
13                TimeStatClassAdapter classAdapter = new TimeStatClassAdapter(cw);
14                cr.accept(classAdapter, ClassReader.SKIP_DEBUG);
15                return cw.toByteArray();
16            }else{
17                System.out.println(className);
18                return classfileBuffer;
19            }
20        }
21
22    },true);
23  }
24 }
```

上述代码第 5 行，加入了类转换器，这里依然使用了匿名内部类的形式定义这个转换器。第 9 行，判断当前加载的类是否为 Account，如果是则进行字节码织入。第 11~15 行使用 TimeStatClassAdapter 对 Account 进行修改。

将 PreMainAddTimeStatAgent 打包成 ja.jar，设置 META-INF/MANIFEST.MF 为：

```
Manifest-Version: 1.0
Premain-Class: geym.zbase.ch11.agent.PreMainAddTimeStatAgent
Can-Redine-Classes: true
Can-Retransform-Classes: true
```

用 Java 虚拟机参数-javaagent:d:/ja.jar 运行上一节中的 RunAccountMain，得到如下输出：

```
agentArgs:null
sun/launcher/LauncherHelper
geym/zbase/ch11/agent/RunAccountMain
java/lang/Void
meet geym.zbase.ch11.aop.timestat.Account
java/lang/InterruptedException
geym/zbase/ch11/aop/timestat/TimeStat
operation....
geym.zbase.ch11.aop.timestat.Account.operation(Unknown Source) spend:10
java/lang/Shutdown
```

```
java/lang/Shutdown$Lock
```

可以看到，使用了 Java Agent 后，已经可以在 main()函数运行时，动态修改 Account 的字节码并使之生效，因此，函数的执行时间也出现在输出中（字体加粗部分）。

注意： 使用 Java Agent 可以在 Class 类型被加载前，进行对类重定义，动态修改指定的 Class 二进制数据并使之生效。这种方式，比基于文件进行字节码织入更为快捷有效。

11.4.3 动态重转换类

在前两节中，使用 Java Agent 方式需要在 Java 程序启动时，加入-javaagent 参数，并且只能在类加载前进行重定义。事实上，Java Agent 也支持在类加载后，进行动态修改。这需要使用 Java Agent 的另外一个方法。

```
public static void agentmain (String agentArgs, Instrumentation inst)
```

函数 agentmain()与 premain()的参数有着同样的含义。但是 agentmain()是在一个 Java Agent 被附加（attach）到 Java 虚拟机上时执行的。当 Java Agent 被 attach 到 Java 虚拟机上，Java 程序的 main()函数一般已经启动，并且程序很可能已经运行了相当长的时间。此时，通过 Instrumentation.retransformClasses()方法，可以使用类转换器重新转换给定的 Class 类型并使之生效。为演示这个功能，请看下面的示例。

【示例 11-23】下面这段代码不断地调用 Account.operation()方法。

```
public class RunLoopAccountMain {
    public static void main(String[] args) {
        Account account = new Account();
        while(true){
            account.operation();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

以上代码持续进行 operation()方法调用，并每次休眠 1 秒钟。不需要加任何 Java 虚拟机参数运行这段代码，程序会持续出现以下输出：

```
operation....
operation....
operation....
```

接着，修改上节中的 PreMainAddTimeStatAgent 类，并增加以下方法：

```
public static void agentmain (String agentArgs, Instrumentation inst) throws
ClassNotFoundException, UnmodifiableClassException{
        System.out.println("Agent Main called");
        System.out.println("agentArgs:"+agentArgs);
        premain(agentArgs,inst);
        inst.retransformClasses(Account.class);
    }
```

上述代码实现了 agentmain()方法，该方法会在函数 Java Agent 被 attach 到虚拟机上时执行。方法 agentmain()委托 premain()方法进行类转换器的注册，并在最后使用了 retransformClasses()要求使用注册的类转换器重转换类 Accout。

将修改后的 PreMainAddTimeStatAgent 类打包为 ja.jar，并修改 META-INF/MANIFEST.MF 为（增加了 Agent-Class）：

```
Manifest-Version: 1.0
Agent-Class: geym.zbase.ch11.agent.PreMainAddTimeStatAgent
Premain-Class: geym.zbase.ch11.agent.PreMainAddTimeStatAgent
Can-Redine-Classes: true
Can-Retransform-Classes: true
```

增加的 Agent-Class 属性表示：当持有 Java Agent 的 jar（这里是 ja.jar）被 attach 到 Java 虚拟机上时，执行该属性指定类的 agentmain()方法（这里就是 PreMainAddTimeStatAgent.agentmain()方法）。

到此，Java Agent 已经准备完毕，最后剩下的问题就是如何将 Java Agent（ja.jar）attach 到正在执行的 RunLoopAccountMain 上。要实现这个功能，需要借助%JAVA_HOME%/lib/tools.jar，该 jar 包提供了一些虚拟机操作的工具类，将该 jar 包加入工程的 ClassPath 中，如图 11.18 所示。

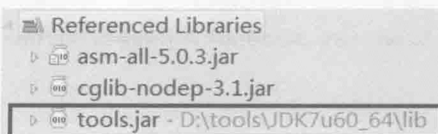


图 11.18　tools. jar 引入

接着，使用 tools.jar 中的工具类，将 ja.jar 附加到目标 Java 进程中，如下代码所示：

```
01 public class AttachToolMain {
02     public static void main(String[] args)
03     throws AttachNotSupportedException, IOException,
04     AgentLoadException, AgentInitializationException {
05         List<VirtualMachineDescriptor> list = VirtualMachine.list();
06         for (VirtualMachineDescriptor vmd : list)
07         {
08             if(vmd.displayName().endsWith("RunLoopAccountMain")){
09                 VirtualMachine virtualmachine = VirtualMachine.attach(vmd.id());
10                 virtualmachine.loadAgent("D:\\ja.jar", "argument for agent");
11                 System.out.println("ok");
12                 virtualmachine.detach();
13             }
14         }
15     }
16 }
```

上述代码的主要功能是将 D:\ja.jar 中的 Java Agent 附加到指定的 Java 程序中。代码第 5 行，获得当前系统中所有的 Java 虚拟机（有点像 jps 命令）。代码第 8 行，限定只对 RunLoopAccountMain 为主函数的虚拟机程序进行操作。代码第 9 行使用 attach()方法，连接上给定的虚拟机（这里就是 RunLoopAccountMain）。代码第 10 行，将准备好的 ja.jar 载入目标虚拟机，并传入了一个参数“argument for agent”。

保持 RunLoopAccountMain 持续运行，并同时运行 AttachToolMain，AttachToolMain 结束后打印出“ok”字样，再去观察 RunLoopAccountMain 的输出，可以看到：

```
operation....
operation....
Agent Main called
agentArgs:argument for agent
agentArgs:argument for agent
meet geym.zbase.ch11.aop.timestat.Account
geym/zbase/ch11/aop/timestat/TimeStat
operation....
geym.zbase.ch11.aop.timestat.Account.operation(Unknown Source) spend:10
operation....
geym.zbase.ch11.aop.timestat.Account.operation(Unknown Source) spend:10
```

由这段输出可知，在 RunLoopAccountMain 加载了 PreMainAddTimeStatAgent 之后，agentmain()函数被调用，接着重新运行了类转换器，并为 Account 类加入了函数调用的计时功能。在类转换结束后，每次 operation()操作后都会输出函数调用的耗时。而这一切都不需要重

启应用，甚至不需要任何额外参数的支持。

11.4.4 有关 Java Agent 的总结

Java Agent 提供了一个很好的方式动态修改类的实现，而 ASM 提供了一套高效的 Class 二进制流创建和修改工具。ASM 可以让 Java Agent 知道如何去修改或者创建一个类，而 Java Agent 可以让 ASM 更好更方便地去进行类的修改。两者的结合，可以大大增加 Java 平台的灵活性。

11.5 与时俱进：动态函数调用

在 JDK 1.7 中引入了一个新的 invoke 指令——invokedynamic，该指令的目的是为了更好地支持 Java 虚拟机平台上的动态语言，以及 Java 8 中的 lambda 表达式。在 Java 8 之前，甚至无法使用 Java 编译器产生 invokedynamic 指令，即使在 JDK 1.7 中已经支持了该指令。随着 invokedynamic 的引入，Java1.7 中还引入了一些新的概念，为了更好地理解 invokedynamic，这里先来介绍一下相关概念。

- **方法句柄（Method Handler）**：方法句柄很像一个方法指针，或者代理。通过方法句柄，就可以调用一个方法。读者应该还记得在 Class 文件的常量池中，有一项常量类型为 CONSTANT_MethodHandle，这就是方法句柄。
- **调用点（CallSite）**：调用点是对方法句柄的封装，通过调用点，可以获得一个方法句柄进行函数调用。使用调用点可以增强方法句柄的表达能力，比如对于可变调用点来说，它绑定的方法句柄是可变的，因此，对同一个调用点而言，其调用函数是可变的。
- **启动方法（BootstrapMethods）**：通过启动方法可以获得一个调用点，获取调用点的目的是为了进行方法绑定和调用。启动方法在 Class 文件的属性中进行描述，读者可以参见 9.2.16 节。
- **方法类型（Method Type）**：用于描述方法的签名，比如方法的参数类型、返回值等。根据方法的类型，可以查找到可用的方法句柄。

11.5.1 方法句柄使用实例

在 JDK 1.7 中，引入了一些新的 API 用来支持方法句柄的使用，这些类主要位于 java.lang.invoke 包中。本小节主要介绍最基础的 4 个类：

- java.lang.invoke.MethodType：MethodType 类提供了一组生成方法类型描述的 API。使

用其静态方法 methodType()，可以根据返回值和参数类型，生成一个 MethodType 对象。

- java.lang.invoke.MethodHandle：表示方法句柄。通过方法句柄的实例，可以使用其 invoke() 方法直接调用指定方法。
- java.lang.invoke.MethodHandles：这是一个工具类，大部分方法是静态方法，用于构造 MethodHandle 实例。
- java.lang.invoke.MethodHandles.Lookup：这是 MethodHandles 的内部类，是一个工具类，用于查找和构造 MethodHandle 实例。

【示例 11-24】下面是一个简单地使用 MethodHandle 进行函数调用的例子。

```
01 package geym.zbase.ch11.inv;
02
03 import java.lang.invoke.MethodHandle;
04 import java.lang.invoke.MethodType;
05 import static java.lang.invoke.MethodHandles.lookup;
06
07 public class SimpleMethodHandle {
08     static class MyPrintln {
09         protected void println(String s) {
10             System.out.println(s);
11         }
12     }
13     public static void main(String[] args) throws Throwable {
14         Object obj = (System.currentTimeMillis() & 1L) == 0L ? System.out : 
new MyPrintln();
15         System.out.println(obj.getClass().toString());
16         getPrintlnMethodHandler(obj).invokeExact("geym");
17     }
18     private static MethodHandle getPrintlnMethodHandler(Object receiver) 
throws Throwable {
19         MethodType mt = MethodType.methodType(void.class, String.class);
20         return lookup().findVirtual(receiver.getClass(), "println", mt).
bindTo(receiver);
21     }
22 }
```

本例中，申明了一个内部类 MyPrintln，它拥有一个 println()方法，这个方法和 System.out 其中一个 println()方法有相同的签名。因此，这两个属于不同类的方法拥有相同的描述和 MethodType。在 main()函数中，第 14 行随机产生一个对象，它可能是 PrintStream 类，也可能是 MyPrintln 类。在第 18 行的 getPrintlnMethodHandler()函数，使用 MethodType.methodType()

方法，构造了一个表示方法签名的 MethodType 实例。这里产生的 MethodType 表示一个返回值为 void，参数为 String 的函数。在第 20 行，使用 MethodHandles.lookup.lookup()得到一个 Lookup 实例，并调用其 findVirtual()方法，根据 MethodType、函数名称和实际的对象类型进行方法查找，得到 MethodHandle 实例，并将调用对象 receiver 作为方法调用的执行者，最后返回这个 MethodHandle 实例。在第 16 行，根据返回的 MethodHandle 实例，调用 println()方法，并传入参数“geym”。无论当前 obj 是什么类型的对象，只要其拥有给定签名和名称的函数，就可以顺利调用到这个方法。

以上就是 MethodHandle 的使用案例。这里还需要说明一点，Lookup 对象进行函数查找时有 3 种方式。

- findStatic()函数：查找一个 static 方法，使用 invokestatic 进行函数调用。
- findVirtual()函数：查找一个虚方法，使用 invokevirtual 指令进行函数调用。
- findSpecial()函数：查找一个特殊的方法，等价于使用 invokespecial 调用，用于访问私有方法、父类的方法。但是对于构造函数的访问需要使用 findConstructor()方法。

由于大部分 Java 函数都是虚函数，因此，对于绝大部分函数调用，使用 findVirtual()方法查找就可以了。对于静态函数则使用 findStatic()方法，对于特殊的一些方法，比如私有方法或者父类方法则使用 findSpecial ()进行查找。

【示例 11-25】下面的代码演示了 findStatic()方法的使用。和 findVirtual()方法不同，静态方法不需要绑定调用对象，因为静态方法没有 this 引用。下面的代码通过 MethodHandle 调用了 Math.sin()函数。

```
public class SimpleStaticMethodHandle{
   public static void main(String[] args) throws Throwable {
      SimpleStaticMethodHandle obj = new SimpleStaticMethodHandle();
      System.out.println(obj.callSin());
   }

   public double callSin() throws Throwable {
      MethodHandle mh = MethodHandles.lookup().findStatic(Math.class, "sin",
            MethodType.methodType(double.class,double.class));
      return (double)mh.invokeExact(Math.PI/2);
   }
}
```

【示例 11-26】下面的例子，使用 findSpecial()调用了类的私有方法。

```
public class SimplePrivateMethodHandle{
    public static void main(String[] args) throws Throwable {
        SimplePrivateMethodHandle obj = new SimplePrivateMethodHandle();
        obj.callToString();
    }

    private void printLine() {
        System.out.println("call private method");
    }

    public void callToString() throws Throwable {
        MethodHandle mh = MethodHandles.lookup().findSpecial(this.getClass(),
"printLine",
                MethodType.methodType(void.class),
this.getClass()).bindTo(this);
        mh.invokeExact();
    }
}
```

【示例 11-27】下面的例子使用 findSpecial()调用了父类的方法，模拟了 super.toString()方法的调用。虽然在 SimpleSuperMethodHandle 类中重载了 toString()方法，但是按照下面所示的调用方式，程序依然会打印“geym.zbase.ch11.inv.SimpleSuperMethodHandle@15e538e”。由于这个方法调用的依然是实例方法，故在最后通过 bindTo()方法绑定到目标实例上。

```
public class SimpleSuperMethodHandle{
  public static void main(String[] args) throws Throwable {
    SimpleSuperMethodHandle obj = new SimpleSuperMethodHandle();
    System.out.println(obj.callToString());
  }

  @Override
  public String toString() {
    return "I am SimpleSuperMethodHandle";
  }

  public String callToString() throws Throwable {
    MethodHandle mh = MethodHandles.lookup().findSpecial(Object.class, "toString",
        MethodType.methodType(String.class), this.getClass()).bindTo(this);
    String a = (String) mh.invokeExact();
    return a;
  }
}
```

注意：findSpecial()的参数，第 1 个为要调用方法的所在类，第 2 个参数为方法名，第 3 个参数为方法类型，第 4 个参数为实际的对象类型。

11.5.2　调用点使用实例

调用点（CallSite）也是 JDK 1.7 内新增的 API，它也在 java.lang.invoke 包中，如图 11.19 所示。调用点用于包装方法句柄，通过一个调用点实例，就可以得到相关的方法句柄，从而实现方法调用。



图 11.19　调用点类层次结构

可以看到，目前，JDK 中支持 3 类调用点，分别是常量调用点（ConstantCallSite），可变调用点（MutableCallSite）和易变调用点（VolatileCallSite）。它们的特点如下。

- 常量调用点：调用目标不可变。一旦它绑定到了目标函数句柄上，就无法再更改。
- 可变调用点：调用目标可变，可以多次绑定到不同的目标函数句柄上，每次只通过同一个调用点，就可以调用不同的函数。
- 易变调用点：由于缓存等原因，可变调用点在多线程环境中，当一个线程更改了可变调用点的目标函数后，其他线程不能保证立即发现这个更改。如果需要保证调用点修改在多线程间的可见性，可以使用易变调用点，它满足 volatile 语义。

以下代码演示了常量调用点的使用：

```
01 public static void constantCallSiteSample() throws Throwable {
02     MethodHandles.Lookup lookup = MethodHandles.lookup();
03     MethodType type = MethodType.methodType(String.class, int.class, int.class);
04     MethodHandle mh = lookup.findVirtual(String.class, "substring", type);
05     ConstantCallSite callSite = new ConstantCallSite(mh);
06     MethodHandle invoker = callSite.dynamicInvoker();
07     String result = (String) invoker.invoke("1234567890", 2, 4);
08     System.out.println("constantCallSiteSample return:"+result);
09 }
```

代码第 5 行，根据 MethodHandle 方法句柄，构造一个常量调用点，常量调用点不可变，一

直指向该 MethodHandle。由第 2~4 行可知，该 MethodHandle 指向 String.substring(int,int)方法。第 6 行，由调用点的 dynamicInvoker()方法取得 MethodHandle，并进行调用。

【示例 11-28】下面的例子展示了可变调用点（MutableCallSite）的使用方式。

```
01 public static void mutableCallSiteSample() throws Throwable {
02     MethodType type = MethodType.methodType(double.class, double.class);
03     MutableCallSite callSite = new MutableCallSite(type);
04     MethodHandle invoker = callSite.dynamicInvoker();
05     MethodHandles.Lookup lookup = MethodHandles.lookup();
06     MethodHandle mhSin = lookup.findStatic(Math.class, "sin", type);
07     MethodHandle mhCos = lookup.findStatic(Math.class, "cos", type);
08     callSite.setTarget(mhSin);
09     double result = (double) invoker.invoke(Math.PI/2);
10     System.out.println("sin(90)="+result);
11     callSite.setTarget(mhCos);
12     result = (double) invoker.invoke(Math.PI/2);
13     System.out.println("cos(90)="+result);
14 }
```

代码第 3 行，根据函数类型（函数签名）构造了可变调用点，根据可变调用点生成 MethodHandle，名为 invoker，但此时，可变调用点尚未完成初始化，没有绑定任何可用函数。接着在第 6、7 行获得可用的方法句柄 Math.sin()和 Math.cos()两个函数。第 8 行设置可变调用点的目标函数为 Math.sin()，并进行函数调用。第 11 行，改变可用调用点的目标函数，并进行调用。程序结果是，通过同一个 MethodHandle 实例 invoker，完成两次不同的函数调用。

11.5.3 反射和方法句柄

读者也许会发现，方法句柄提供的函数查找和访问功能，在反射中不是早已提供了吗？不错，使用反射也可以实现类似的功能，不同的是，反射是在 Java 语言层面进行方法调用模拟；而方法句柄则是在 Java 字节码层面进行函数调用。因此，方法句柄的执行效率要高于反射。

【示例 11-29】使用下面的代码对反射调用和方法句柄调用进行测试。

```
public class RelectionMain {
 public static final int COUNT = 1000000;
 int i = 0;

 public void method() {
     i++;
 }
```

```
08
09  public static void callByHandler() throws Throwable {
10       RelectionMain instance = new RelectionMain();
11       MethodType mt = MethodType.methodType(void.class);
12       MethodHandle mh = lookup().findVirtual(instance.getClass(), "method",
mt).bindTo(instance);
13       long b = System.currentTimeMillis();
14       for (int i = 0; i < COUNT; i++) {
15           mh.invokeExact();
16       }
17       long e = System.currentTimeMillis();
18       System.out.println(e - b);
19  }
20
21  public static void callByReflection() throws Throwable {
22       RelectionMain instance = new RelectionMain();
23       Method m = instance.getClass().getMethod("method");
24       long b = System.currentTimeMillis();
25       for (int i = 0; i < COUNT; i++) {
26           m.invoke(instance);
27       }
28       long e = System.currentTimeMillis();
29       System.out.println(e - b);
30  }
31
32  public static void main(String[] args) throws Throwable {
33       callByHandler();
34       callByHandler();
35       callByReflection();
36       callByReflection();
37  }
38 }
```

调用的目标函数都是第 5 行的 method()方法。函数 callByHandler()使用方法句柄进行函数调用，函数 callByReflection()使用反射进行调用。两者都用 method()方法执行相同的调用次数，并统计时间。主函数 main()中，分别对两个测试函数进行两次调用：第 1 次调用属于热身，希望相关的 JIT 编译等进行完成，第 2 次调用才是要关心的性能统计数据。

在笔者计算机上，使用如下参数运行，得到如下输出：

```
运行参数：-Xint
```

```
callByHandler spend:282
callByHandler spend:263
callByReflection spend:444
callByReflection spend:433

运行参数：-Xcomp -XX:-BackgroundCompilation
callByHandler spend:52
callByHandler spend:50
callByReflection spend:121
callByReflection spend:67
```

这里，只关心每一种类型的第 2 次输出（加粗部分）。可以看到，对于纯解释型函数调用，反射的性能只能达到方法句柄的一半，而对于 JIT 编译后的函数调用，反射的性能有较大提升，但仍然比方法句柄略差。

11.5.4 指令 invokedynamic 使用实例

在了解了方法句柄和调用点的基础上，终于可以让主角 invokedynamic 指令登场亮相了。指令 invokedynamic 接收两个字节作为其操作数，根据操作数，invokedynamic 会计算出常量池索引，该常量池入口为 CONSTANT_InvokeDynamic 结构，该结构用于指向一个引导方法（BootstrapMethods）和方法的签名。引导方法会指向常量池中的 CONSTANT_MethodHandle 项，表示方法的调用位置。CONSTANT_MethodHandle 入口会具体指向 CONSTANT_Methodref 或者 CONSTANT_Fieldref 结构。上述包括引导方法、CONSTANT_MethodHandle、CONSTANT_InvokeDynamic 在内的属性或者常量池类型，在第 9 章中有详细描述，读者可以参考相关结构。

通过对 InvokeDynamic 结构的解析，最终会找到引导方法，引导方法会返回一个 CallSite 调用点，虚拟机在执行引导方法后，就能得到目标的调用点，并取得目标函数的 MethodHandle，从而完成函数调用。

如图 11.20 所示，invokedynamic 指令会找到 Class 文件中的 CONSTANT_InvokeDynamic 常量，并解析出对应的引导方法 BootstrapMethods。在引导方法结构中，又会引用常量池中的 MethodHandle 方法句柄，该方法句柄表示引导方法，通过这个 MethodHandle 又能在 Class 文件中找到 CONSTANT_Methodref，从而进行引导方法的调用。引导方法会返回 CallSite 调用点，系统通过调用点可以找到最终需要调用的方法句柄，就可以执行目标方法了。由此可见，动态调用的函数派发绑定是由引导方法决定的。这也就是为什么把它称为引导方法的原因。

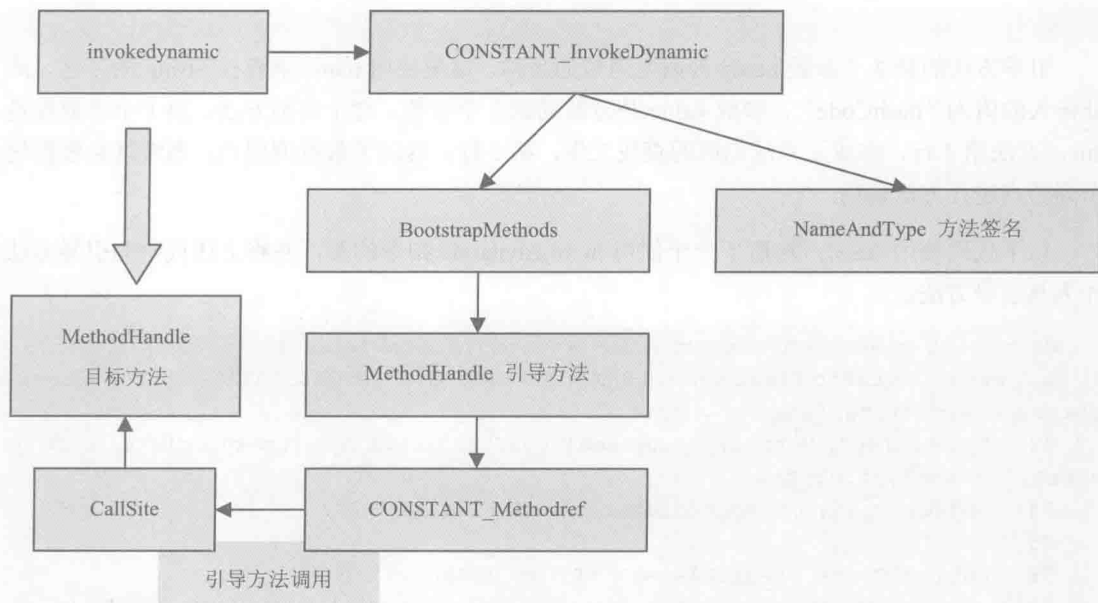


图 11.20　invokedynamic 指令流程图

目前，指令 invokedynamic 完全是为动态语言准备的。在 JDK 1.7 以及之前版本，Java 尚不具备很强的动态性，因此无法通过 Java 编译器产生 invokedynamic 指令。但在 JDK 1.8 之后，由于函数式编程的引入，Java 的动态性大大增强，在 Java 函数式 API 调用中，编译器就能产生 invokedynamic 指令。此外，由各种动态语言自己的编译器（比如 groovy 等），也可能产生 invokedynamic 调用。

虽然在 JDK 1.8 之前，无法直接由 Java 编译器产生 invokedynamic 指令，但在前文中，已有不少篇幅介绍 ASM 框架的使用。在这里，将使用 ASM 产生包含 invokedynamic 指令的函数，并演示其结构。本例使用 invokedynamic 来调用 String.hashCode()方法。

为了使用 invokedynamic，必须先建立一个引导方法，代码如下：

```
public class DynBootStrap {
        public static CallSite bootstrap(Lookup lookup, String name,
MethodType type, Object value) throws Exception {
       System.out.println("bootstrap called,name="+name);
          MethodHandle mh = lookup.findVirtual(value.getClass(), name,
MethodType.methodType(int.class)).bindTo(value);
        return new ConstantCallSite(mh);
    }
```

```
07 }
```

引导方法的第 2 个参数 name 为调用函数的名称，这里使用 name 来查找 String 的方法，此处传入的值为“hashCode”。参数 value 是方法的第 1 个参数，对于实例方法，第 1 个参数就是 this。方法第 4 行，完成了方法句柄的查找工作，第 5 行，返回了常量调用点，系统就会根据这个调用点进行方法调用。

以下代码使用 ASM，构造了一个使用 invokedynamic 指令的类，并将上述代码的引导方法作为其引导方法。

```
public class DynInvokerSample extends ClassLoader {
private static final Handle BSM = new Handle(H_INVOKESTATIC, DynBootStrap.class.getName().replace('.', '/'),
 "bootstrap", MethodType.methodType(CallSite.class, Lookup.class, String.class, MethodType.class,
 Object.class).toMethodDescriptorString());

 public Class createClass() throws IOException {
     ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
    cw.visit(V1_7, ACC_PUBLIC | ACC_SUPER, "DynInvokerSampleMain", null, "java/lang/Object", null);
     Method m = Method.getMethod("void <init> ()");
     GeneratorAdapter mg = new GeneratorAdapter(ACC_PUBLIC, m, null, null, cw);
     mg.loadThis();
     mg.invokeConstructor(Type.getType(Object.class), m);
     mg.returnValue();
     mg.endMethod();

    MethodVisitor mv = cw.visitMethod(ACC_PUBLIC | ACC_STATIC, "run", "()V", null, null);
     mv.visitCode();

     mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
     mv.visitInvokeDynamicInsn("hashCode", "()I", BSM, "geym");
     mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(I)V");

     mv.visitInsn(RETURN);
     mv.visitMaxs(0, 0);
     mv.visitEnd();
     cw.visitEnd();
```

```
27        byte[] bytes = cw.toByteArray();
28
29        FileOutputStream fos = new FileOutputStream(new File("D:/
DynInvokerSampleMain.class"));
30        fos.write(bytes);
31        return this.defineClass("DynInvokerSampleMain", bytes, 0, bytes.length);
32    }
33
34    public static void main(String[] args) throws IOException, InstantiationException,
IllegalAccessException,
35            IllegalArgumentException, InvocationTargetException,
NoSuchMethodException, SecurityException {
36        DynInvokerSample me = new DynInvokerSample();
37        Object obj = me.createClass().newInstance();
38        obj.getClass().getMethod("run").invoke(null);
39        System.out.println("geym".hashCode());
40    }
41 }
```

方法第 2~4 行，申明了引导方法，引导方法将使用 invokstatic 进行调用。第 8~14 行，创建了类 DynInvokerSampleMain，并为其产生了默认的构造函数。第 16 行，创建了 run()方法。第 20 行，使用 invokedynamic 指令，将在字符串“geym”上，查找并调用名为 hashCode 的函数，函数签名为“()I”，即不接收参数，返回值为 int。传入的 BSM 为在第 2~4 行申明的引导方法。第 21 行，输出 invokedynamic 运行后的结果。第 29 行，将新生成的 Class 写入文件 DynInvokerSampleMain.class。第 31 行，完成类 DynInvokerSampleMain 的定义和加载。第 38 行，调用了 DynInvokerSampleMain.run()方法，打印“geym”的 hashCode()返回。第 39 行，直接调用"geym".hashCode()。运行以上代码输出如下：

```
bootstrap called,name=hashCode
3169394
3169394
```

使用 javap 工具分析上述代码产生的 DynInvokerSampleMain.class 文件，如图 11.21 所示。文件中有两处结构和动态调用有关：

- 一处是 run() 方法字节码部分使用了 invokedynamic 指令，该指令一方面指向了 CONSTANT_InvokeDynamic 结构，另一方面指向了第一个引导方法。
- 另一处就是引导方法属性，它主要定位了方法句柄的位置，并根据常量池之间的引用关系，可以一直追溯到 DynBootStrap. bootstrap()方法。

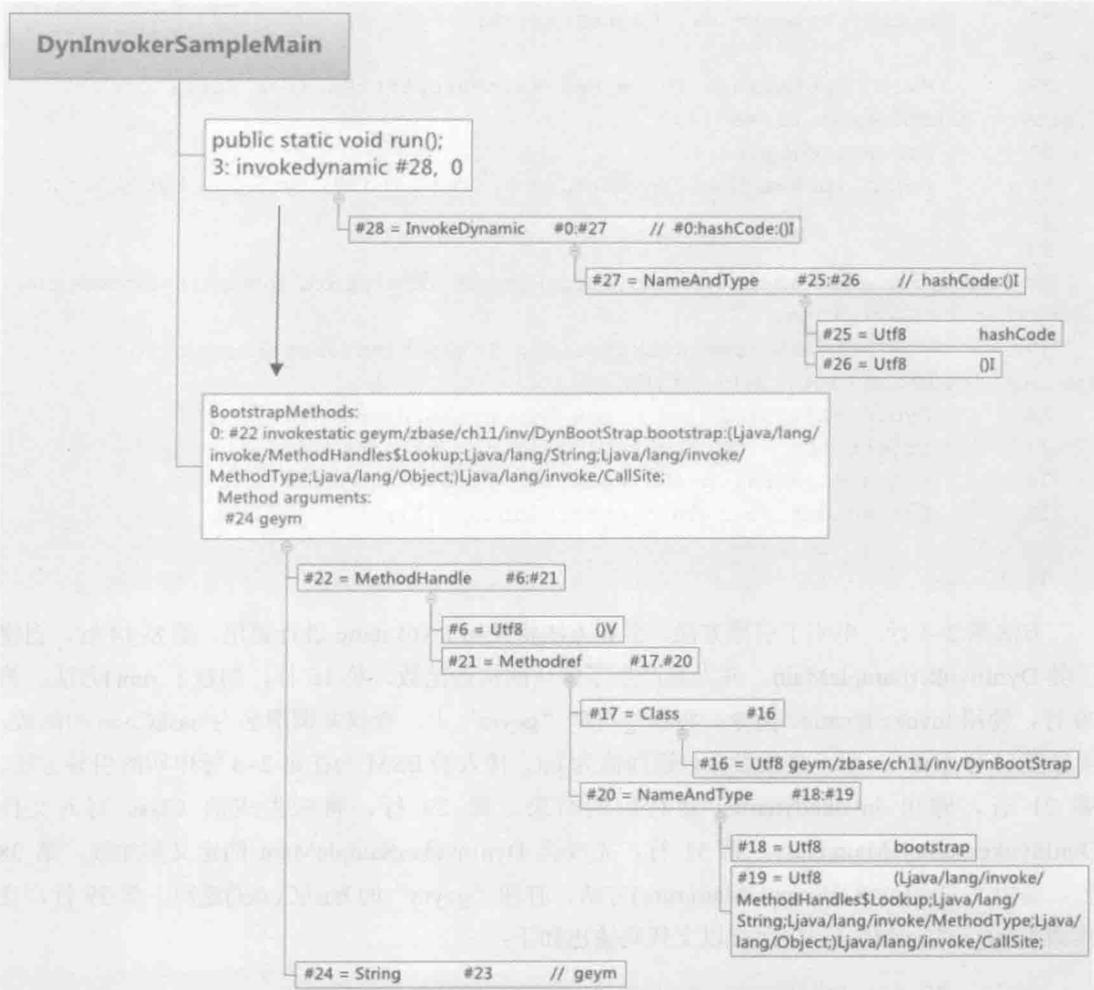


图 11.21 invokedynamic 实例结构图

11.6 跑得再快点：静态编译优化

当使用 javac 把 Java 源码转为字节码时，编译器会有一些优化以获得更好的性能。目前，对于执行的字节码会从两处进行优化：

第一，就是使用 javac 编译时；

第二，就是通过 JIT（Just-In-Time）即时编译，在运行时。

目前，大量的优化工作都围绕着 JIT 展开，比如方法内联、栈上替换等。将优化工作从 javac 前端移到后端的好处是非常明显的，这样，所有基于 Java 平台的语言都能共享这种优化带来的好处。将大量的优化只放置于 javac 前端，那么只有 Java 语言可以利用这种优化方式。但即便如此，开发人员也必须要了解一些 javac 的常用优化方法。

11.6.1　编译时计算

如果在程序中出现了计算表达式，如果表达式的值能够在编译时确定，那么表达式的计算会提前到编译阶段，而不是在运行时计算。

【示例 11-30】很多时候，为了增强代码的可读性，往往不会把最终的数值写在代码中，通常倾向于把计算过程写在代码里。比如下面代码：

```
for(int i=0;i<60*60*24*1000;i++){
    //do sth.
}
```

循环次数为 60*60*24*1000 次，通常这个表达式可能是用来计算天时分秒的乘积。看到这段代码，可能会让人产生一种怀疑，是不是这个计算每次循环都要进行一次呢？如果是的话，是不是更应该写成：

```
for(int i=0;i< 86400000;i++){
    //do sth.
}
```

或者一定要保留计算表达式的话：

```
int count=60*60*24*1000;
for(int i=0;i< count;i++){
    //do sth.
}
```

读者也许会认为，上述代码先计算了表达式乘积，并保留这个值，以避免每次循环都重复计算。

实际上，后两段代码的担心是多余的，因为在编译的时候，对于给定的表达式会自动计算并给出结果。本例中第一段代码生成的字节码如下：

```
#20 = Integer           86400000
0: iconst_0
```

```
1: istore_1
2: goto        8
5: iinc        1, 1
8: iload_1
9: ldc         #20              // int 86400000
11: if_icmplt   5
14: return
```

可以看到，用于控制循环次数上限的整数在字节码中并非经过计算得来，而是保存在常量池中，并直接使用，其作用是用来判定是否可以继续循环。可见，对于常量表达式，可以大胆地使用而无需担心影响系统性能。

【示例 11-31】另一个常用的例子是字符串连接。有时候，如果一个字符串很长，通常会倾向于使用“+”号连接。由于字符串是不可变的对象，读者也许会认为使用类似 A+B 的方式连接字符串时，需要 3 个对象，即 A、B 和 AB。下面再来看一个例子。

```
public static void createString(){
    String info1="select * from test";
    String info2="select * "+"from test";
    String info3="select * ".concat("from test");
    System.out.println(info1==info2);
    System.out.println(info1==info3);
    System.out.println(info2==info3);
System.out.println(info2==info3.intern());
}
```

上述代码中，info1 是直接定义的字符，info2 使用“+”号连接，生成字面量等于 info1 的字符串，info3 使用 String.concat()方法做连接生成。如果执行以上代码，输出如下：

```
true
false
false
true
```

可以看到，info1 和 info2 是指向了同一个对象引用，而 info3 则是指向了不同的对象引用，但是 info3 的常量池引用地址就是 info2。这说明 info3 是被实实在在构造出来的新的 String 对象，而 info2 的“+”号运算并未在运行时进行，否则也应该有新对象产生。查看它的部分字节码：

```
0:   ldc     #24; //String select * from test
2:   astore_0
3:   ldc     #24; //String select * from test
5:   astore_1
```

```
   6:  ldc     #26; //String select *
   8:  ldc     #28; //String from test
   10:  invokevirtual   #30; //Method java/lang/String.concat:(Ljava/lang/String;)
Ljava/lang/String;
   13:  astore_2
```

上述字节码中，第 2 行表示将常量池第 24 项存入第 0 个局部变量（info1），第 5 行表示将常量池第 24 项存入第 1 个局部变量（info2）。这里就解释了为什么程序会有这样的输出，因为在编译时，字符串连接已经完成。而对于后续的 concat()函数，则没有这种优化，第 10 行的 invokevirtual 调用，就是说明了 info3 是在运行时被创建的。

因此，对于常量字符串连接，不能担心多写几个“+”号就会影响系统性能、多占用内存等，因为这些都会在编译器进行计算。

11.6.2　变量字符串的连接

上一节介绍了编译器对常量字符串的优化，那么对于变量字符串连接，是否也有类似的优化方案呢？答案是肯定的。

【示例 11-32】下面这段代码是非常常用的字符串操作方式。

```
public static void addString(String str1,String str2){
    String str3=str1+str2;
}
```

变量 str1 和 str2 通过“+”号进行字符串连接。对于常量，会在运行时进行计算，而对于变量，运行时显然无法进行计算，此时，编译器会为字符串的操作插入 StringBuilder 来进行。笔者使用 jad（Java 的反编译器）对上述代码生成的 Class 文件进行反编译，得到：

```
String str3 = (new StringBuilder(String.valueOf(str1))).append(str2).toString();
```

可以清楚地看到，字符串连接都被转为了更可取的 StringBuilder 操作，避免了每次字符串操作都生成新的对象，因此，读者大可不必担心这类操作带来太大的性能问题。但是如果能在编码时留意这些问题，依然是大有好处的。

【示例 11-33】下面的代码在循环中进行字符串连接。

```
public static void addString2(String ...str1){
    String str3="";
    for(String str:str1){
        str3+=str1;
    }
}
```

根据上述的经验，读者应该能想到，这些字符串操作，都会被编译成更高效的 StringBuilder。使用 jad 反编译上述生成的 Class 文件，得到：

```
public static void addString2(String str1[])
{
    String str3 = "";
    String as[];
    int j = (as = str1).length;
    for(int i = 0; i < j; i++)
    {
        String str = as[i];
        str3 = (new StringBuilder(String.valueOf(str3))).append(str1).toString();
    }
}
```

从反编译代码可知，虽然使用了 StringBuilder 进行字符串连接，但是，优化的结果是为每次字符串连接都生成一个 StringBuilder 实例，这显然也是一种不够聪明的做法。更好的做法是在循环外建立 StringBuilder 实例，在循环内进行字符串累加操作。

由此可见，对于变量字符串累加，编译器会做适量优化，但是并无法得到最优的解决方案，依然需要在开发时注意这些问题。

11.6.3 基于常量的条件语句裁剪

在介绍本节内容前，笔者先讲述一个真实的开发案例。众所周知，对于大型项目的构建过程是非常耗时的，以笔者维护的 Java 项目为例，完全编译和构建需要花费 30 分钟到 1 小时左右。因此，在正式发布版本之前，大家都是非常谨慎的，生怕出错后重新编译项目费时费力。但是，难免疏忽，有时候就是会由于这样那样的原因，需要重新编译。为了节省时间，有时候就会使用“塞包”的方法，将需要更新的类单独编译，然后更新到已经编译完成的 jar 包中，而不是去重新编译整个系统（当然，一般情况下并不推荐这么做）。

【示例 11-34】有一次，笔者发现类似下面代码定义的类需要修改。

```
public class FinalFlag {
    public static final boolean flag=true;
}
```

修改的内容是 flag 常量应该被置为 false，而不是 true。因为在系统中有不少地方以下面类似的方式引用这个常量。

```
public void checkflag(){
    if(FinalFlag.flag){
        System.out.println("flag is true");
    }else{
        System.out.println("flag is false");
    }
}
```

为了避免重新编译整个系统带来的麻烦，笔者想当然地将 FinalFlag 中的 flag 置为 false，单独编译 FinalFlag，并将新的 Class 文件更新到 jar 中。接着更新整个系统并进行测试，结果发现引用到 FinalFlag.flag 的代码点，并没有进入期望的逻辑，让人纠结半天。

其实，导致这个问题的原因很简单，如果使用 javap 获得上述 checkflag()方法的字节码，可以看到：

```
   0:   getstatic      #36; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc    #59; //String flag is true
   5:   invokevirtual #61; //Method java/io/PrintStream.println:(Ljava/lang/
String;)V
   8:   return
```

这是不是令人吃惊的编译结果？在实际执行的字节码中，没有去获取 FinalFlag.flag，没有进行条件判断跳转，只是简单的打印了“String flag is true”。因此，无论如何更新 FinalFlag.flag，只要引用这个变量的类不做更新，程序都不会有任何变化。这就是因为编译器会对常量做特殊的优化，由于 final 常量是不可变的，因此，任何逻辑都可以在编译时就确定，不需要的逻辑自然可以裁剪，因此就有了这么一个问题。

解决这个问题的办法就是做一次全面的编译，让 final 的更新影响到每一个使用到的类中。

11.6.4　switch 语句的优化

在前面的章节中提到，对于 switch 语句，编译器会产生两种字节码指令：tableswitch 和 lookupswitch。其中指令 tableswitch 的效率高于 lookupswitch，但 tableswitch 的结构决定了它只能处理 case 情况是连续的数值，而 lookupswitch 可以处理不连续的情况。一般来说，总是希望可以尽可能地使用 tableswitch，而不是 lookupswitch。

【示例 11-35】以下面代码为例。

```
01 public void switch(int i){
02     switch(i){
03         case 1:break;
```

```
        case 2:break;
        case 5:break;
        default:
            System.out.println("");
    }
}
```

也许不少读者会认为，因为case值是不连续的，所以编译器会使用低效的lookupswitch指令处理这个switch，每次都遍历查找所有的可能性。

但如果用javap查看生成的字节码，会发现：

```
0:   iload_1
1:   tableswitch{ //1 to 5
            1: 36;
            2: 39;
            3: 45;
            4: 45;
            5: 42;
            default: 45 }
36:  goto    53
39:  goto    53
42:  goto    53
45:  getstatic     #36; //Field java/lang/System.out:Ljava/io/PrintStream;
48:  ldc     #58; //String
50:  invokevirtual  #60; //Method java/io/PrintStream.println:(Ljava/lang/
String;)V
53:  return
```

可以看到，为了使用tableswitch，编译器将离散的数据进行了填充，其中case3、4都跳转到了default，符合代码的语义。但如果离散空间很大，就会要求插入过多的中间数据，此时编译器依然会使用lookupswitch指令。

11.7 提高虚拟机的执行效率：JIT 及其相关参数

JIT（Just-In-Time）编译器是Java虚拟机的执行机制的性能保证。由于Java的字节码是解释执行的，因此其效率很低。在Java发展历史中，有两套解释执行器：古老的字节码解释器、现在被广泛使用的模板解释器。字节码解释器在执行时，通过纯软件代码模拟字节码的执行，效率非常低下。相比之下，模板解释器将每一条字节码和一个模板函数相关联，而模板函数中

直接产生这条字节码执行时的机器码，从而很大程度上提高了解释器的性能。但即便如此，仅凭借解释器，虚拟机的执行效率依然很低，为了解决这个问题，虚拟机平台支持一种叫做即时编译的技术。

即时编译的目的是避免函数被解释执行，而是将整个函数体编译成机器码，每次函数执行时，只执行编译后的机器码即可，这种方式可以使执行效率大幅度提升。

11.7.1　开启 JIT 编译

Java 虚拟机有 3 种执行方式，分别是解释执行、混合模式和编译执行，默认情况下处于混合模式中。使用命令行 java –version 可以查看虚拟机的执行模式：

```
C:\Users\Administrator>java -version
java version "1.7.0_13"
Java(TM) SE Runtime Environment (build 1.7.0_13-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, mixed mode)
```

上面输出的“mixed mode”就表示混合模式。在混合模式中，部分函数会被解释执行，部分可能被编译执行。虚拟机决定函数是否需要编译执行的依据是判断该函数是否为热点代码。如果函数的调用频率很高，被反复使用，那么就会被认为是热点，热点代码就会被编译执行。

解释执行模式表示全部代码均解释执行，不做任何 JIT 编译，可以使用参数-Xint 来开启解释执行模式：

```
C:\Users\Administrator>java -Xint -version
java version "1.7.0_13"
Java(TM) SE Runtime Environment (build 1.7.0_13-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, interpreted mode)
```

编译执行模式和解释执行模式相反，对于所有的函数，无论是否是热点代码，都会被编译执行，使用参数-Xcomp 可以设置为编译模式：

```
C:\Users\Administrator>java -Xcomp -version
java version "1.7.0_13"
Java(TM) SE Runtime Environment (build 1.7.0_13-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, compiled mode)
```

一般来说，编译模式的执行效率会远远高于解释模式。

【示例 11-36】下面的代码不停计算圆周率的数值，并给出了运行的耗时。

```
public static double calcPi(){
```

```
    double re=0;
    for(int i=1;i<10000;i++){
        re+=((i&1)==0?-1:1)*1.0/(2*i-1);
    }
    return re*4;
}
public static void main(String[] args) {
    long b=System.currentTimeMillis();
    for(int i=0;i<10000;i++)
        calcPi();
    long e=System.currentTimeMillis();
    System.out.println("spend:"+(e-b)+"ms");
}
```

使用虚拟机参数-Xint 运行以上代码，输出：

```
spend:2794ms
```

使用虚拟机参数-Xcomp 运行以上代码，输出：

```
spend:1082ms
```

很明显，在本例中使用编译运行要比解释运行快大约 3 倍。

11.7.2 JIT 编译阈值

默认情况下，JIT 使用混合模式运行（指定参数为-Xmixed），在混合模式时，只会对热点代码进行即时编译。对于是否为热点代码，虚拟机内有一个阈值进行判断，当函数调用次数超过这个阈值时，就被认为是热点代码，进行即时编译。在 Client 模式下，这个阈值为 1500 次，在 Server 模式下阈值为 10000 次。使用参数-XX:CompileThreshold 可以设置这个阈值，使用参数-XX:+PrintCompilation 可以打印出即时编译的日志。

【示例 11-37】下面代码调用了 500 次 met()函数，在默认情况下，met()函数不会被编译，但是通过设置-XX:CompileThreshold 为 500，则 met()函数就会成为热点代码，被编译为机器码。

```
public static void met(){
    int a=0,b=0;
    b=a+b;
}
public static void main(String[] args) throws InterruptedException {
    for(int i=0;i<500;i++){
        met();
```

```
    }
    Thread.sleep(1000);
}
```

以参数-XX:CompileThreshold=500 -XX:+PrintCompilation 运行以上代码，部分输出如下：

```
68   19  s        java.lang.StringBuffer::append (8 bytes)
69   20           java.io.Win32FileSystem::normalize (231 bytes)
70   21           geym.zbase.ch11.jit.JITSimpleTest::met (9 bytes)
```

可以看到，函数 met()已经被编译为机器码。

11.7.3　多级编译器

Java 虚拟机中拥有客户端编译器和服务端编译器两种编译系统，一般称为 C1 和 C2 编译器。当使用-client 参数时，虚拟机会使用 C1 编译器，使用-server 参数时，会使用 C2 编译器。C1 编译器的特点是编译速度快，C2 的特点是会做更多的编译时优化，因此编译时间会长于 C1，但是编译后的代码质量会高于 C1。为了使 C1 和 C2 在编译速度和执行效率之间取得一个平衡，虚拟机支持一种叫做多级编译的策略。多级编译将编译的层次分为 5 级：

- 0 级（解释执行）：采用解释执行，不采集性能监控数据。
- 1 级（简单的 C1 编译）：采用 C1 编译器，进行最简单的快速编译，根据需要采集性能数据。
- 2 级（有限的 C1 编译）：采用 C1 编译器，进行更多的优化编译，可能会根据第 1 级采集的性能统计数据，进一步优化编译代码。
- 3 级（完全 C1 编译）：完全使用 C1 编译器的所有功能，会采集性能数据进行优化。
- 4 级（C2 编译）：完全使用 C2 进行编译，进行完全的优化。

要使用多级编译器可以使用参数-XX: +TieredCompilation 打开多级编译器的策略。如果使用该参数，那么虚拟机必须使用-server 模式启动，如果使用-client 模式启动，那么分级模式依然不会开启。

【示例 11-38】下面的代码频繁调用了 WriterService. service()函数。通过不同的参数配置，来学习一下多级编译器的使用。

```
public class WriterMain {
    public static void main(String[] args) throws InterruptedException {
        long b=System.currentTimeMillis();
        WriterService ws=new WriterService();
        for(int i=0;i<20000000;i++){
```

```
            ws.service();
        }
        long e=System.currentTimeMillis();
        System.out.println("spend:"+(e-b));
        ws=null;
        System.gc();
        Thread.sleep(5000);
    }
}

public class WriterService {
    public void service(){
        DBWriter writer=new DBWriter();
        writer.write();
    }
}

public class DBWriter {
    public void write() {
        "DBWriter".toCharArray();
    }
}
```

使用参数-XX:+PrintCompilation -server –Xcomp 运行上述代码，部分输出如下：

```
    862  382    b     geym.zbase.ch11.jit.deopt.WriterMain::main (74 bytes)
    863  382          geym.zbase.ch11.jit.deopt.WriterMain::main (74 bytes)
made not entrant
    912  397    b     geym.zbase.ch11.jit.deopt.WriterService::<init> (5 bytes)
    913  398    b     geym.zbase.ch11.jit.deopt.WriterService::service (13 bytes)
    914  398          geym.zbase.ch11.jit.deopt.WriterService::service (13 bytes)
made not entrant
    915  400    b     geym.zbase.ch11.jit.deopt.DBWriter::<init> (5 bytes)
    915  401    b     geym.zbase.ch11.jit.deopt.DBWriter::write (7 bytes)
    918  403    b     geym.zbase.ch11.jit.deopt.WriterService::service (13 bytes)
    919  404 %  b     geym.zbase.ch11.jit.deopt.WriterMain::main @ 18 (74 bytes)
    1143  404 %       geym.zbase.ch11.jit.deopt.WriterMain::main @ -2 (74 bytes)
made not entrant
```

可以看到在 WriterService.service()和 DBWriter.writer()等函数都被编译成了本地代码。其中有些函数后标注有 made not entrant 标记，表示该函数编译结果被标记为不可再用，后续可能会从内存中移除，表示废弃。这种情况是因为该函数已经不需要再使用（比如类被垃圾收集器回

收，其方法代码自然不会再被使用），或者编译器产生了一个更优的编译结果，需要将旧数据清除。

注意： made not entrant 状态只是表示新的代码调用不能使用这块编译结果，但可能存在正在使用该段代码块的程序，故还不能完全从系统中清理。如果代码块已经没有被使用，并处于 made not entrant 状态，一旦被清理线程发现，代码块会被进一步标记为 made zombie 状态（僵尸状态），此时，代码库将被彻底清除。

使用参数-XX:+PrintCompilation -server –Xcomp -XX:+TieredCompilation 运行上述代码，部分输出如下：

```
   50  287    b  3   geym.zbase.ch11.jit.deopt.WriterMain::main (74 bytes)
   57  298    b  3   geym.zbase.ch11.jit.deopt.WriterService::<init> (5 bytes)
   57  299    b  3   geym.zbase.ch11.jit.deopt.WriterService::service (13 bytes)
   57  300    b  3   geym.zbase.ch11.jit.deopt.DBWriter::<init> (5 bytes)
   57  301    b  3   geym.zbase.ch11.jit.deopt.DBWriter::write (7 bytes)
   59  302    b  4   geym.zbase.ch11.jit.deopt.WriterService::service (13 bytes)
   62  299       3   geym.zbase.ch11.jit.deopt.WriterService::service (13 bytes)
made not entrant
   62  303    b  4   geym.zbase.ch11.jit.deopt.DBWriter::<init> (5 bytes)
   63  300       3   geym.zbase.ch11.jit.deopt.DBWriter::<init> (5 bytes) made not
entrant
   63  304    b  4   geym.zbase.ch11.jit.deopt.DBWriter::write (7 bytes)
   63  301       3   geym.zbase.ch11.jit.deopt.DBWriter::write (7 bytes)  made not
entrant
   66  309 %  b  4   geym.zbase.ch11.jit.deopt.WriterMain::main @ 18 (74 bytes)
   70  310    b  4   geym.zbase.ch11.jit.deopt.WriterMain::main (74 bytes)
   73  287       3   geym.zbase.ch11.jit.deopt.WriterMain::main (74 bytes) made
not entrant
   77  309 %     4   geym.zbase.ch11.jit.deopt.WriterMain::main @ -2 (74 bytes)
made not entrant
```

可以看到，在使用-XX:+TieredCompilation 之后，JIT 的日志发生了变化，在输出的结果中多了一列，增加的列表示编译的级别。该日志的每一列分别如下。

- 时间戳：系统启动到编译完成时的毫秒数。
- 即时编译 ID：编译任务的内部 ID，一般是一个自增的值。
- 属性：描述代码状态的 5 个属性。
  - %：是一个 OSR（栈上替换）。
  - s：这是一个同步方法。

    ◦ !：方法有异常处理快。
    ◦ b：阻塞模式编译。
    ◦ n：是本地方法的一个包装。
- 编译级别：0~4 级编译。级别越高编译生成的机器码质量越好，相对的，编译耗时也越长。
- 方法名：被编译方法的名称。
- 方法大小：被编译方法的大小，这个大小是指 Java 的字节码大小，不是生成的本地机器码大小。

从这段编译日志中不难发现，在时间戳第 57 毫秒，301 号编译中，DBWriter.writer()被第一次编译为机器码。编译级别是 3。但在第 63 毫秒处，该 301 号编译被标记位 made not entrant，即不再允许产生新的调用，而与此同时，第 63 毫秒处的 304 号编译将 DBWriter.writer()方法重新编译，并使用编译级别 4。这就是一个典型的层次编译策略进行编译级别的提升，后一次编译产生了更优质的代码。

11.7.4 OSR 栈上替换

一般来说，一次函数调用要么使用解释执行，要么使用即时编译后的机器码执行。从解释执行切换到机器码执行，是在这个函数两次调用之间产生的，即一个函数的前一次函数调用发生时，尚未准备好编译后版本，则会进行解释执行，而下一次调用发生时，发现其编译版本已经准备好，那么就会使用编译后的版本执行。这种情况覆盖了绝大部分场合，但是，不适合那些调用次数不多，但是方法体内包含大量循环的函数，比如类似如下函数：

```
public static void main(String[] args) throws InterruptedException {
    WriterService ws=new WriterService();
    for(int i=0;i<20000000;i++){
        ws.service();
    }
}
```

方法 main()被执行的次数只有一次（或者其他被调用次数明显不多的函数），但是方法体内有一个循环次数很多的循环体，这种情况下，由于 main()函数不可能满足编译条件，永远得不到编译，但是从实际运行的角度来说，循环代码被大量反复执行，是有编译价值的。为了应对这种情况。编译器会记录循环的次数，一旦循环次数达到一定的阈值（一般在 1 万次左右），就会认为这段循环代码也是热点代码，并触发即时编译。

编译完成后，由于 main()方法很少被执行，甚至可能永远不会再执行了。因此，虚拟机需

要一个切入点来动态将正在执行的循环体代码替换为编译后版本。这种不等待函数运行结束，在循环体内就将代码替换为编译版本的技术，叫做 OSR（On Stack Relapcement）栈上替换。如图 11.22 所示，左侧表示普通的方法编译，每次在方法入口进行判断，右侧表示栈上替换，在方法入口时并不检查（方法调用次数很少），而在每次循环开始时，判断是否有编译版本可供使用。

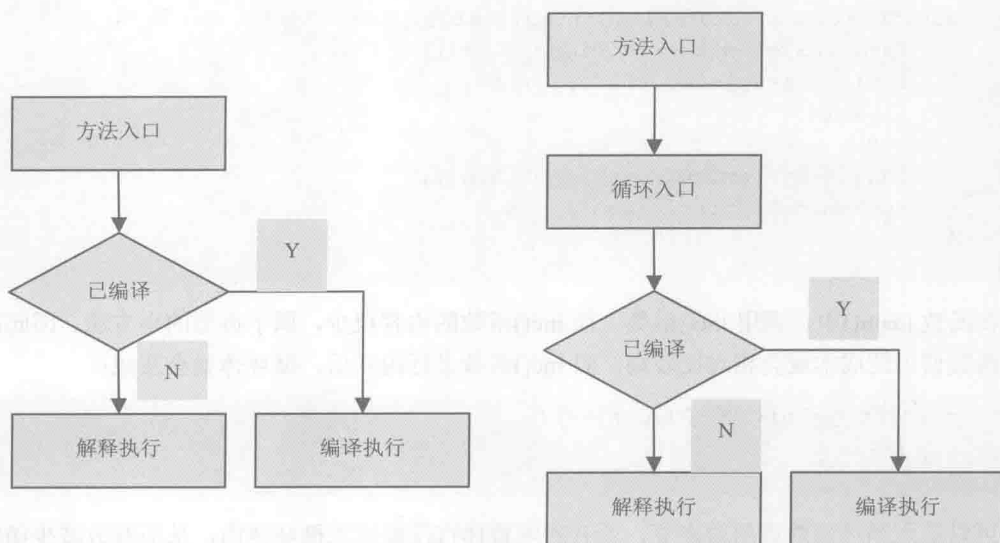


图 11.22　OSR 和普通方法编译区别

在上一节的实例中，main()函数已经使用了 OSR 进行编译和运行。

```
66  309 %  b  4      geym.zbase.ch11.jit.deopt.WriterMain::main @ 18 (74 bytes)
```

上述日志中，%表示 OSR。

注意：严格来说，并不是通过循环次数和循环入口来进行 OSR 判断的，而是使用一种叫做回边（back-edge）计数器和回边指令来进行判断的。所谓回边，就是字节码指令中，向后跳转的指令。当然，大部分情况下，这种向后跳转的指令是由循环产生的。

11.7.5　方法内联

方法内联是一种有效的优化手段，它可以减少方法调用的次数，从而提高系统性能。JIT 编译器默认会进行方法内联优化。

【示例 11-39】下面的例子说明了什么是方法内联。

```
public class InLineMain {
    static int i=0;
    public static void inc(){
        i++;
    }
    public static void main(String[] args) {
        long b=System.currentTimeMillis();
        for(int j=0;j<100000000;j++){
            inc();
        }
        long e=System.currentTimeMillis();
        System.out.println(e-b);
    }
}
```

在函数 main()中，调用 inc()函数。而 inc()函数的内容很少，属于典型的小方法。因此产生一次函数调用的成本就会相对比较高，对 inc()函数进行内联后，循环体就会变成：

```
    for(int j=0;j<100000000;j++){
        i++;
    }
```

可以看到循环函数调用被剥离，原有的函数体直接被嵌入循环体内，从而有效减少函数调用次数。对于这种小方法效果是极其明显的，使用-XX:+Inline 参数可以打开内联优化。使用参数：-Xcomp -server -XX:+Inline 运行示例中的 InLineMain.main()，输出（耗时）：

```
7
```

使用-Xcomp -server -XX:-Inline 参数关闭内联，再次运行，输出（耗时）：

```
236
```

由此可见，内联的优化效果是非常显著的。但是内联会增大系统执行代码的体积，因此对于大的方法体，使用内联也需要谨慎，虚拟机提供了一些参数来控制对于多大的代码允许进行内联，比如-XX:FreqInlineSize 参数，可以控制热点方法进行内联的体积上限，但凡方法体积大于给定值的，都不会进行内联优化。

11.7.6 设置代码缓存大小

字节码被编译为机器码后，得到的结果需要在内存中保存，以便下次函数调用时可以直接使用。存放这些代码的内存区域称为代码缓存（Code Cache）。一旦代码缓存区域被用完，虚

拟机并不会像堆或者永久区那样暴力地直接抛出内存溢出错误，而是简单地停止 JIT 编译，并保持系统继续运行。系统停止 JIT 编译后，后续未编译的代码全部以解释方式运行，故系统性能会受到影响。代码缓存空间的清理工作也是在系统 GC 时完成的。

设置代码缓存区间的大小可以使用参数-XX:ReservedCodeCacheSize 指定。一般在 32 位的 client 模式下，这个值为 32M。读者可以根据自己应用的需要设定这个值。

【示例 11-40】下面一个例子显示了代码缓存区间的大小设置，以及代码缓存区间的回收情况。读者需要知道，代码缓存区间设置越大，那么可以保存的被编译的方法就越多。而一旦代码缓存区间耗尽，JIT 编译器停止工作，那么就再也无法启动 JIT 编译了。

```
public class CodeCacheJit  implements Opcodes {
  public static void createAndCall(String hellostr,String classname)
throws IllegalAccessException, IllegalArgumentException, InvocationTargetException,
SecurityException{
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS|ClassWriter.
COMPUTE_FRAMES);
    cw.visit(V1_7, ACC_PUBLIC, classname, null, "java/lang/Object", null);
    MethodVisitor mw = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null,
        null);
    mw.visitVarInsn(ALOAD, 0);
    mw.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "<init>", "()V");
    mw.visitInsn(RETURN);
    mw.visitMaxs(0, 0);
    mw.visitEnd();
    mw = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",
        "([Ljava/lang/String;)V", null, null);
    mw.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
        "Ljava/io/PrintStream;");
    mw.visitLdcInsn(hellostr);
    mw.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println",
        "(Ljava/lang/String;)V");
    mw.visitInsn(RETURN);
    mw.visitMaxs(0,0);
    mw.visitEnd();
    byte[] code = cw.toByteArray();

    JitClassLoader loader = new JitClassLoader();
    Method m;
  try {
      m = ClassLoader.class.getDeclaredMethod("defineClass", String.class,
```

```
byte[].class,int.class,int.class);
28          m.setAccessible(true);
29          Class exampleClass=(Class) m.invoke(loader, classname, code, 0, code.
length);
30          exampleClass.getMethods()[0].invoke(null, new Object[] { null });
31      } catch (NoSuchMethodException e) {
32          e.printStackTrace();
33      }
34
35  }
36  public static void main(final String args[]) throws Exception {
37    for(int i=0;i<Integer.MAX_VALUE;i++){
38          createAndCall("hello,world"+i,"Ex"+i);
39          //停止 jit 后再 gc ，jit 不会启用了
40          if(i>=2500){
41             System.gc();
42          }
43      }
44  }
45 }
```

上述代码在 main()函数中反复调用 createAndCall()函数，这个函数每次新建一个类，并调用这个类的 main()方法。程序第 40 行，在调用 2500 次以后，进行显式的 GC 操作。

使用以下参数运行这段代码：

```
-XX:+PrintCompilation -XX:ReservedCodeCacheSize=5M -Xcomp
```

得到程序部分输出如下：

```
hello,world2041
  1143 2777    b       Ex2042::main (9 bytes)
hello,world2042
  1143 2778    b       Ex2043::main (9 bytes)
hello,world2043
hello,world2044
hello,world2045
hello,world2046
…
hello,world2571
hello,world2572
```

可以看到，在 2041 次调用后，JIT 编译被停止，不再有任何 JIT 编译日志信息输出。而在

2500 次调用后，系统调用显式 GC 后，并没有任何效果。

将程序第 40 行改为：

```
if(i>=2000){
```

以相同的参数运行以上代码，可以看到部分输出（输出量太大，故截取小部分）：

```
hello,world1270
    869 2006    b       Ex1271::main (9 bytes)
......
hello,world1999
   1125 2735    b       Ex2000::main (9 bytes)
hello,world2000
   1125 2736    b       java.lang.System::gc (7 bytes)
   1125 2737     n      java.lang.Runtime::gc (native)
   1138 2739    b       Ex2001::main (9 bytes)
hello,world2001
   1148 2740    b       Ex2002::main (9 bytes)
   1148 1063           (method)   made zombie
   1148 1061      hello,world2002     (method)
   made zombie
   1148 1058           (method)   made zombie
   1157 2454           (method)   made zombie
   1157 2453           (method)   made zombie
.......
  19278 4729           (method)   made zombie
hello,world4002
  19289 4741    b       Ex4003::main (9 bytes)
hello,world4003
  19298 4742    b       Ex4004::main (9 bytes)
hello,world4004
```

可以看到在 2000 次调用前，没有任何代码缓存空间被回收，在 2000 次调用后，System.gc()运行，并开始回收代码缓存空间，大量函数被标记为 made zombie，并被释放。由于垃圾回收的存在，无效的代码总是可以被释放，故代码缓存空间未被耗尽，并最终持续执行到 4000 多次调用，依然正常工作。

注意：一旦代码缓存耗尽，JIT 就会停止，并且在整个虚拟机的生命周期中，不会再启动了。要扩大代码缓存区间，可以使用-XX:ReservedCodeCacheSize 参数设置。

如图 11.23 所示，显示了在 JConsole 里观察代码缓存的使用情况。

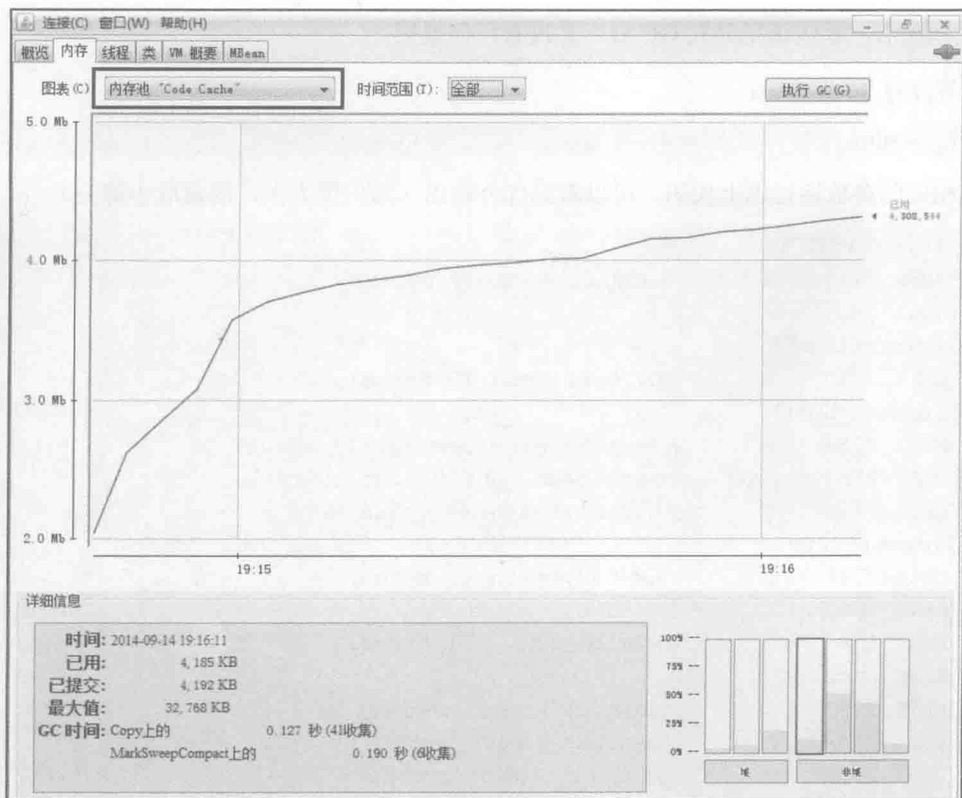


图 11.23 代码缓存使用率

11.8 小结

本章主要介绍了 Java 虚拟机的字节码执行机制。首先介绍了 Java 虚拟机的主要指令集，接着以这些指令集为基础，学习如何使用 ASM 库进行动态字节码生成、修改等相关的开发工作。为了更好地使用这种技术，本章还介绍了 Java Agent 机制。

此外，本章还简单介绍了 Javac 的静态编译以及 JIT 动态编译这两项基础性的模块，以帮助读者进一步理解 Java 虚拟机的运行机制。